

Probabilistic Machine Learning

Adaptive Computation and Machine Learning

Francis Bach, editor

A complete list of titles can be found online at <https://mitpress.mit.edu/search-result-list/?series=adaptive-computation-and-machine-learning-series>.

Probabilistic Machine Learning

Advanced Topics

Kevin P. Murphy

The MIT Press
Cambridge, Massachusetts
London, England

© 2023 Kevin P. Murphy

This work is subject to a Creative Commons CC-BY-NC-ND license.

Subject to such license, all rights are reserved.



The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Murphy, Kevin P., author.

Title: Probabilistic machine learning : advanced topics / Kevin P. Murphy.

Description: Cambridge, Massachusetts : The MIT Press, [2023] | Series:
Adaptive computation and machine learning series | Includes
bibliographical references and index.

Identifiers: LCCN 2022045222 (print) | LCCN 2022045223 (ebook) | ISBN
9780262048439 (hardcover) | ISBN 9780262376006 (epub) | ISBN
9780262375993 (pdf)

Subjects: LCSH: Machine learning. | Probabilities.

Classification: LCC Q325.5 .M873 2023 (print) | LCC Q325.5 (ebook) | DDC
006.3/1015192–dc23/eng20230111

LC record available at <https://lccn.loc.gov/2022045222>

LC ebook record available at <https://lccn.loc.gov/2022045223>

10 9 8 7 6 5 4 3 2 1

This book is dedicated to my wife Margaret,
who has been the love of my life for 20+ years.

Brief Contents

1 Introduction 1

I Fundamentals 3

- 2 Probability 5
- 3 Statistics 63
- 4 Graphical models 143
- 5 Information theory 217
- 6 Optimization 255

II Inference 337

- 7 Inference algorithms: an overview 339
- 8 Gaussian filtering and smoothing 353
- 9 Message passing algorithms 395
- 10 Variational inference 433
- 11 Monte Carlo methods 477
- 12 Markov chain Monte Carlo 493
- 13 Sequential Monte Carlo 537

III Prediction 567

- 14 Predictive models: an overview 569
- 15 Generalized linear models 583
- 16 Deep neural networks 623
- 17 Bayesian neural networks 639
- 18 Gaussian processes 673
- 19 Beyond the iid assumption 727

IV Generation 763

20 Generative models: an overview	765
21 Variational autoencoders	781
22 Autoregressive models	811
23 Normalizing flows	819
24 Energy-based models	839
25 Diffusion models	857
26 Generative adversarial networks	883

V Discovery 915

27 Discovery methods: an overview	917
28 Latent factor models	919
29 State-space models	969
30 Graph learning	1033
31 Nonparametric Bayesian models	1037
32 Representation learning	1039
33 Interpretability	1063

VI Action 1093

34 Decision making under uncertainty	1095
35 Reinforcement learning	1135
36 Causality	1173

Contents

Preface xxix

1 Introduction 1

I Fundamentals 3

2 Probability 5

2.1	Introduction	5
2.1.1	Probability space	5
2.1.2	Discrete random variables	5
2.1.3	Continuous random variables	6
2.1.4	Probability axioms	7
2.1.5	Conditional probability	7
2.1.6	Bayes' rule	8
2.2	Some common probability distributions	8
2.2.1	Discrete distributions	9
2.2.2	Continuous distributions on \mathbb{R}	10
2.2.3	Continuous distributions on \mathbb{R}^+	13
2.2.4	Continuous distributions on $[0, 1]$	17
2.2.5	Multivariate continuous distributions	17
2.3	Gaussian joint distributions	22
2.3.1	The multivariate normal	22
2.3.2	Linear Gaussian systems	28
2.3.3	A general calculus for linear Gaussian systems	30
2.4	The exponential family	33
2.4.1	Definition	34
2.4.2	Examples	34
2.4.3	Log partition function is cumulant generating function	39
2.4.4	Canonical (natural) vs mean (moment) parameters	41
2.4.5	MLE for the exponential family	42
2.4.6	Exponential dispersion family	43
2.4.7	Maximum entropy derivation of the exponential family	43
2.5	Transformations of random variables	44
2.5.1	Invertible transformations (bijections)	44
2.5.2	Monte Carlo approximation	45
2.5.3	Probability integral transform	45
2.6	Markov chains	46
2.6.1	Parameterization	47
2.6.2	Application: language modeling	49

2.6.3	Parameter estimation	49
2.6.4	Stationary distribution of a Markov chain	51
2.7	Divergence measures between probability distributions	55
2.7.1	<i>f</i> -divergence	55
2.7.2	Integral probability metrics	57
2.7.3	Maximum mean discrepancy (MMD)	58
2.7.4	Total variation distance	61
2.7.5	Density ratio estimation using binary classifiers	61
3 Statistics	63	
3.1	Introduction	63
3.2	Bayesian statistics	63
3.2.1	Tossing coins	64
3.2.2	Modeling more complex data	70
3.2.3	Selecting the prior	71
3.2.4	Computational issues	71
3.2.5	Exchangeability and de Finetti's theorem	71
3.3	Frequentist statistics	72
3.3.1	Sampling distributions	72
3.3.2	Bootstrap approximation of the sampling distribution	73
3.3.3	Asymptotic normality of the sampling distribution of the MLE	74
3.3.4	Fisher information matrix	75
3.3.5	Counterintuitive properties of frequentist statistics	79
3.3.6	Why isn't everyone a Bayesian?	82
3.4	Conjugate priors	83
3.4.1	The binomial model	83
3.4.2	The multinomial model	83
3.4.3	The univariate Gaussian model	85
3.4.4	The multivariate Gaussian model	90
3.4.5	The exponential family model	96
3.4.6	Beyond conjugate priors	98
3.5	Noninformative priors	102
3.5.1	Maximum entropy priors	102
3.5.2	Jeffreys priors	103
3.5.3	Invariant priors	106
3.5.4	Reference priors	107
3.6	Hierarchical priors	107
3.6.1	A hierarchical binomial model	108
3.6.2	A hierarchical Gaussian model	110
3.6.3	Hierarchical conditional models	113
3.7	Empirical Bayes	114
3.7.1	EB for the hierarchical binomial model	114
3.7.2	EB for the hierarchical Gaussian model	115
3.7.3	EB for Markov models (n-gram smoothing)	116
3.7.4	EB for non-conjugate models	118
3.8	Model selection	118
3.8.1	Bayesian model selection	119
3.8.2	Bayes model averaging	121
3.8.3	Estimating the marginal likelihood	121
3.8.4	Connection between cross validation and marginal likelihood	122
3.8.5	Conditional marginal likelihood	123
3.8.6	Bayesian leave-one-out (LOO) estimate	124
3.8.7	Information criteria	125
3.9	Model checking	127
3.9.1	Posterior predictive checks	128
3.9.2	Bayesian p-values	130

3.10	Hypothesis testing	131
3.10.1	Frequentist approach	131
3.10.2	Bayesian approach	131
3.10.3	Common statistical tests correspond to inference in linear models	136
3.11	Missing data	141
4	Graphical models	143
4.1	Introduction	143
4.2	Directed graphical models (Bayes nets)	143
4.2.1	Representing the joint distribution	143
4.2.2	Examples	144
4.2.3	Gaussian Bayes nets	148
4.2.4	Conditional independence properties	149
4.2.5	Generation (sampling)	154
4.2.6	Inference	155
4.2.7	Learning	155
4.2.8	Plate notation	161
4.3	Undirected graphical models (Markov random fields)	164
4.3.1	Representing the joint distribution	165
4.3.2	Fully visible MRFs (Ising, Potts, Hopfield, etc.)	166
4.3.3	MRFs with latent variables (Boltzmann machines, etc.)	172
4.3.4	Maximum entropy models	174
4.3.5	Gaussian MRFs	177
4.3.6	Conditional independence properties	179
4.3.7	Generation (sampling)	181
4.3.8	Inference	181
4.3.9	Learning	182
4.4	Conditional random fields (CRFs)	185
4.4.1	1d CRFs	186
4.4.2	2d CRFs	189
4.4.3	Parameter estimation	192
4.4.4	Other approaches to structured prediction	193
4.5	Comparing directed and undirected PGMs	193
4.5.1	CI properties	193
4.5.2	Converting between a directed and undirected model	195
4.5.3	Conditional directed vs undirected PGMs and the label bias problem	196
4.5.4	Combining directed and undirected graphs	197
4.5.5	Comparing directed and undirected Gaussian PGMs	199
4.6	PGM extensions	201
4.6.1	Factor graphs	201
4.6.2	Probabilistic circuits	204
4.6.3	Directed relational PGMs	205
4.6.4	Undirected relational PGMs	207
4.6.5	Open-universe probability models	210
4.6.6	Programs as probability models	210
4.7	Structural causal models	211
4.7.1	Example: causal impact of education on wealth	212
4.7.2	Structural equation models	213
4.7.3	Do operator and augmented DAGs	213
4.7.4	Counterfactuals	214
5	Information theory	217
5.1	KL divergence	217
5.1.1	Desiderata	218
5.1.2	The KL divergence uniquely satisfies the desiderata	219
5.1.3	Thinking about KL	222
5.1.4	Minimizing KL	223

5.1.5	Properties of KL	226
5.1.6	KL divergence and MLE	228
5.1.7	KL divergence and Bayesian inference	229
5.1.8	KL divergence and exponential families	230
5.1.9	Approximating KL divergence using the Fisher information matrix	231
5.1.10	Bregman divergence	231
5.2	Entropy	232
5.2.1	Definition	233
5.2.2	Differential entropy for continuous random variables	233
5.2.3	Typical sets	234
5.2.4	Cross entropy and perplexity	235
5.3	Mutual information	236
5.3.1	Definition	236
5.3.2	Interpretation	237
5.3.3	Data processing inequality	237
5.3.4	Sufficient statistics	238
5.3.5	Multivariate mutual information	239
5.3.6	Variational bounds on mutual information	242
5.3.7	Relevance networks	244
5.4	Data compression (source coding)	245
5.4.1	Lossless compression	245
5.4.2	Lossy compression and the rate-distortion tradeoff	246
5.4.3	Bits back coding	248
5.5	Error-correcting codes (channel coding)	249
5.6	The information bottleneck	250
5.6.1	Vanilla IB	250
5.6.2	Variational IB	251
5.6.3	Conditional entropy bottleneck	252
6	Optimization	255
6.1	Introduction	255
6.2	Automatic differentiation	255
6.2.1	Differentiation in functional form	255
6.2.2	Differentiating chains, circuits, and programs	260
6.3	Stochastic optimization	265
6.3.1	Stochastic gradient descent	265
6.3.2	SGD for optimizing a finite-sum objective	267
6.3.3	SGD for optimizing the parameters of a distribution	267
6.3.4	Score function estimator (REINFORCE)	268
6.3.5	Reparameterization trick	269
6.3.6	Gumbel softmax trick	271
6.3.7	Stochastic computation graphs	272
6.3.8	Straight-through estimator	273
6.4	Natural gradient descent	273
6.4.1	Defining the natural gradient	274
6.4.2	Interpretations of NGD	275
6.4.3	Benefits of NGD	276
6.4.4	Approximating the natural gradient	276
6.4.5	Natural gradients for the exponential family	278
6.5	Bound optimization (MM) algorithms	281
6.5.1	The general algorithm	281
6.5.2	Example: logistic regression	282
6.5.3	The EM algorithm	283
6.5.4	Example: EM for an MVN with missing data	285
6.5.5	Example: robust linear regression using Student likelihood	287
6.5.6	Extensions to EM	289

6.6	Bayesian optimization	291
6.6.1	Sequential model-based optimization	292
6.6.2	Surrogate functions	292
6.6.3	Acquisition functions	294
6.6.4	Other issues	297
6.7	Derivative-free optimization	298
6.7.1	Local search	298
6.7.2	Simulated annealing	301
6.7.3	Evolutionary algorithms	301
6.7.4	Estimation of distribution (EDA) algorithms	304
6.7.5	Cross-entropy method	306
6.7.6	Evolutionary strategies	306
6.8	Optimal transport	307
6.8.1	Warm-up: matching optimally two families of points	308
6.8.2	From optimal matchings to Kantorovich and Monge formulations	308
6.8.3	Solving optimal transport	311
6.9	Submodular optimization	316
6.9.1	Intuition, examples, and background	316
6.9.2	Submodular basic definitions	318
6.9.3	Example submodular functions	320
6.9.4	Submodular optimization	322
6.9.5	Applications of submodularity in machine learning and AI	327
6.9.6	Sketching, coresets, distillation, and data subset and feature selection	327
6.9.7	Combinatorial information functions	331
6.9.8	Clustering, data partitioning, and parallel machine learning	332
6.9.9	Active and semi-supervised learning	332
6.9.10	Probabilistic modeling	333
6.9.11	Structured norms and loss functions	335
6.9.12	Conclusions	335

II Inference 337

7	Inference algorithms: an overview	339
7.1	Introduction	339
7.2	Common inference patterns	340
7.2.1	Global latents	340
7.2.2	Local latents	341
7.2.3	Global and local latents	341
7.3	Exact inference algorithms	342
7.4	Approximate inference algorithms	342
7.4.1	The MAP approximation and its problems	343
7.4.2	Grid approximation	344
7.4.3	Laplace (quadratic) approximation	345
7.4.4	Variational inference	346
7.4.5	Markov chain Monte Carlo (MCMC)	348
7.4.6	Sequential Monte Carlo	349
7.4.7	Challenging posteriors	350
7.5	Evaluating approximate inference algorithms	350
8	Gaussian filtering and smoothing	353
8.1	Introduction	353
8.1.1	Inferential goals	353
8.1.2	Bayesian filtering equations	355
8.1.3	Bayesian smoothing equations	356
8.1.4	The Gaussian ansatz	357
8.2	Inference for linear-Gaussian SSMs	357

8.2.1	Examples	358
8.2.2	The Kalman filter	359
8.2.3	The Kalman (RTS) smoother	364
8.2.4	Information form filtering and smoothing	366
8.3	Inference based on local linearization	369
8.3.1	Taylor series expansion	369
8.3.2	The extended Kalman filter (EKF)	370
8.3.3	The extended Kalman smoother (EKS)	373
8.4	Inference based on the unscented transform	373
8.4.1	The unscented transform	375
8.4.2	The unscented Kalman filter (UKF)	376
8.4.3	The unscented Kalman smoother (UKS)	376
8.5	Other variants of the Kalman filter	377
8.5.1	General Gaussian filtering	377
8.5.2	Conditional moment Gaussian filtering	380
8.5.3	Iterated filters and smoothers	381
8.5.4	Ensemble Kalman filter	382
8.5.5	Robust Kalman filters	384
8.5.6	Dual EKF	384
8.5.7	Normalizing flow KFs	384
8.6	Assumed density filtering	385
8.6.1	Connection with Gaussian filtering	386
8.6.2	ADF for SLDS (Gaussian sum filter)	387
8.6.3	ADF for online logistic regression	388
8.6.4	ADF for online DNNs	391
8.7	Other inference methods for SSMs	391
8.7.1	Grid-based approximations	392
8.7.2	Expectation propagation	392
8.7.3	Variational inference	393
8.7.4	MCMC	393
8.7.5	Particle filtering	394
9	Message passing algorithms	395
9.1	Introduction	395
9.2	Belief propagation on chains	395
9.2.1	Hidden Markov Models	396
9.2.2	The forwards algorithm	397
9.2.3	The forwards-backwards algorithm	398
9.2.4	Forwards filtering backwards smoothing	401
9.2.5	Time and space complexity	402
9.2.6	The Viterbi algorithm	403
9.2.7	Forwards filtering backwards sampling	406
9.3	Belief propagation on trees	406
9.3.1	Directed vs undirected trees	406
9.3.2	Sum-product algorithm	408
9.3.3	Max-product algorithm	409
9.4	Loopy belief propagation	411
9.4.1	Loopy BP for pairwise undirected graphs	412
9.4.2	Loopy BP for factor graphs	412
9.4.3	Gaussian belief propagation	413
9.4.4	Convergence	415
9.4.5	Accuracy	417
9.4.6	Generalized belief propagation	418
9.4.7	Convex BP	418
9.4.8	Application: error correcting codes	418
9.4.9	Application: affinity propagation	420

9.4.10	Emulating BP with graph neural nets	421
9.5	The variable elimination (VE) algorithm	422
9.5.1	Derivation of the algorithm	422
9.5.2	Computational complexity of VE	424
9.5.3	Picking a good elimination order	426
9.5.4	Computational complexity of exact inference	426
9.5.5	Drawbacks of VE	427
9.6	The junction tree algorithm (JTA)	428
9.7	Inference as optimization	429
9.7.1	Inference as backpropagation	429
9.7.2	Perturb and MAP	430
10	Variational inference	433
10.1	Introduction	433
10.1.1	The variational objective	433
10.1.2	Form of the variational posterior	435
10.1.3	Parameter estimation using variational EM	436
10.1.4	Stochastic VI	438
10.1.5	Amortized VI	438
10.1.6	Semi-amortized inference	439
10.2	Gradient-based VI	439
10.2.1	Reparameterized VI	440
10.2.2	Automatic differentiation VI	446
10.2.3	Blackbox variational inference	448
10.3	Coordinate ascent VI	449
10.3.1	Derivation of CAVI algorithm	450
10.3.2	Example: CAVI for the Ising model	452
10.3.3	Variational Bayes	453
10.3.4	Example: VB for a univariate Gaussian	454
10.3.5	Variational Bayes EM	457
10.3.6	Example: VBEM for a GMM	458
10.3.7	Variational message passing (VMP)	464
10.3.8	Autoconj	465
10.4	More accurate variational posteriors	465
10.4.1	Structured mean field	465
10.4.2	Hierarchical (auxiliary variable) posteriors	465
10.4.3	Normalizing flow posteriors	466
10.4.4	Implicit posteriors	466
10.4.5	Combining VI with MCMC inference	466
10.5	Tighter bounds	467
10.5.1	Multi-sample ELBO (IWAE bound)	467
10.5.2	The thermodynamic variational objective (TVO)	468
10.5.3	Minimizing the evidence upper bound	468
10.6	Wake-sleep algorithm	469
10.6.1	Wake phase	469
10.6.2	Sleep phase	470
10.6.3	Daydream phase	471
10.6.4	Summary of algorithm	471
10.7	Expectation propagation (EP)	472
10.7.1	Algorithm	472
10.7.2	Example	474
10.7.3	EP as generalized ADF	474
10.7.4	Optimization issues	475
10.7.5	Power EP and α -divergence	475
10.7.6	Stochastic EP	475
11	Monte Carlo methods	477

11.1	Introduction	477
11.2	Monte Carlo integration	477
11.2.1	Example: estimating π by Monte Carlo integration	478
11.2.2	Accuracy of Monte Carlo integration	478
11.3	Generating random samples from simple distributions	480
11.3.1	Sampling using the inverse cdf	480
11.3.2	Sampling from a Gaussian (Box-Muller method)	481
11.4	Rejection sampling	481
11.4.1	Basic idea	482
11.4.2	Example	483
11.4.3	Adaptive rejection sampling	483
11.4.4	Rejection sampling in high dimensions	484
11.5	Importance sampling	484
11.5.1	Direct importance sampling	485
11.5.2	Self-normalized importance sampling	485
11.5.3	Choosing the proposal	486
11.5.4	Annealed importance sampling (AIS)	486
11.6	Controlling Monte Carlo variance	488
11.6.1	Common random numbers	488
11.6.2	Rao-Blackwellization	488
11.6.3	Control variates	489
11.6.4	Antithetic sampling	490
11.6.5	Quasi-Monte Carlo (QMC)	491
12	Markov chain Monte Carlo	493
12.1	Introduction	493
12.2	Metropolis-Hastings algorithm	494
12.2.1	Basic idea	494
12.2.2	Why MH works	495
12.2.3	Proposal distributions	496
12.2.4	Initialization	498
12.3	Gibbs sampling	499
12.3.1	Basic idea	499
12.3.2	Gibbs sampling is a special case of MH	499
12.3.3	Example: Gibbs sampling for Ising models	500
12.3.4	Example: Gibbs sampling for Potts models	502
12.3.5	Example: Gibbs sampling for GMMs	502
12.3.6	Metropolis within Gibbs	504
12.3.7	Blocked Gibbs sampling	504
12.3.8	Collapsed Gibbs sampling	505
12.4	Auxiliary variable MCMC	507
12.4.1	Slice sampling	507
12.4.2	Swendsen-Wang	509
12.5	Hamiltonian Monte Carlo (HMC)	510
12.5.1	Hamiltonian mechanics	511
12.5.2	Integrating Hamilton's equations	511
12.5.3	The HMC algorithm	513
12.5.4	Tuning HMC	514
12.5.5	Riemann manifold HMC	515
12.5.6	Langevin Monte Carlo (MALA)	515
12.5.7	Connection between SGD and Langevin sampling	516
12.5.8	Applying HMC to constrained parameters	517
12.5.9	Speeding up HMC	518
12.6	MCMC convergence	518
12.6.1	Mixing rates of Markov chains	519
12.6.2	Practical convergence diagnostics	520

12.6.3	Effective sample size	523
12.6.4	Improving speed of convergence	525
12.6.5	Non-centered parameterizations and Neal's funnel	525
12.7	Stochastic gradient MCMC	526
12.7.1	Stochastic gradient Langevin dynamics (SGLD)	527
12.7.2	Preconditioning	527
12.7.3	Reducing the variance of the gradient estimate	528
12.7.4	SG-HMC	529
12.7.5	Underdamped Langevin dynamics	529
12.8	Reversible jump (transdimensional) MCMC	530
12.8.1	Basic idea	531
12.8.2	Example	531
12.8.3	Discussion	533
12.9	Annealing methods	533
12.9.1	Simulated annealing	533
12.9.2	Parallel tempering	536
13	Sequential Monte Carlo	537
13.1	Introduction	537
13.1.1	Problem statement	537
13.1.2	Particle filtering for state-space models	537
13.1.3	SMC samplers for static parameter estimation	539
13.2	Particle filtering	539
13.2.1	Importance sampling	539
13.2.2	Sequential importance sampling	541
13.2.3	Sequential importance sampling with resampling	542
13.2.4	Resampling methods	545
13.2.5	Adaptive resampling	547
13.3	Proposal distributions	547
13.3.1	Locally optimal proposal	548
13.3.2	Proposals based on the extended and unscented Kalman filter	549
13.3.3	Proposals based on the Laplace approximation	549
13.3.4	Proposals based on SMC (nested SMC)	551
13.4	Rao-Blackwellized particle filtering (RBPF)	551
13.4.1	Mixture of Kalman filters	551
13.4.2	Example: tracking a maneuvering object	553
13.4.3	Example: FastSLAM	554
13.5	Extensions of the particle filter	557
13.6	SMC samplers	557
13.6.1	Ingredients of an SMC sampler	558
13.6.2	Likelihood tempering (geometric path)	559
13.6.3	Data tempering	561
13.6.4	Sampling rare events and extrema	562
13.6.5	SMC-ABC and likelihood-free inference	563
13.6.6	SMC ²	563
13.6.7	Variational filtering SMC	563
13.6.8	Variational smoothing SMC	564
III	Prediction	567
14	Predictive models: an overview	569
14.1	Introduction	569
14.1.1	Types of model	569
14.1.2	Model fitting using ERM, MLE, and MAP	570
14.1.3	Model fitting using Bayes, VI, and generalized Bayes	571

14.2	Evaluating predictive models	572
14.2.1	Proper scoring rules	572
14.2.2	Calibration	572
14.2.3	Beyond evaluating marginal probabilities	576
14.3	Conformal prediction	579
14.3.1	Conformalizing classification	581
14.3.2	Conformalizing regression	581
15	Generalized linear models	583
15.1	Introduction	583
15.1.1	Some popular GLMs	583
15.1.2	GLMs with noncanonical link functions	586
15.1.3	Maximum likelihood estimation	587
15.1.4	Bayesian inference	587
15.2	Linear regression	588
15.2.1	Ordinary least squares	588
15.2.2	Conjugate priors	589
15.2.3	Uninformative priors	591
15.2.4	Informative priors	593
15.2.5	Spike and slab prior	595
15.2.6	Laplace prior (Bayesian lasso)	596
15.2.7	Horseshoe prior	597
15.2.8	Automatic relevancy determination	598
15.2.9	Multivariate linear regression	600
15.3	Logistic regression	602
15.3.1	Binary logistic regression	602
15.3.2	Multinomial logistic regression	603
15.3.3	Dealing with class imbalance and the long tail	604
15.3.4	Parameter priors	604
15.3.5	Laplace approximation to the posterior	605
15.3.6	Approximating the posterior predictive distribution	607
15.3.7	MCMC inference	609
15.3.8	Other approximate inference methods	610
15.3.9	Case study: is Berkeley admissions biased against women?	611
15.4	Probit regression	613
15.4.1	Latent variable interpretation	613
15.4.2	Maximum likelihood estimation	614
15.4.3	Bayesian inference	616
15.4.4	Ordinal probit regression	616
15.4.5	Multinomial probit models	617
15.5	Multilevel (hierarchical) GLMs	617
15.5.1	Generalized linear mixed models (GLMMs)	618
15.5.2	Example: radon regression	618
16	Deep neural networks	623
16.1	Introduction	623
16.2	Building blocks of differentiable circuits	623
16.2.1	Linear layers	624
16.2.2	Nonlinearities	624
16.2.3	Convolutional layers	625
16.2.4	Residual (skip) connections	626
16.2.5	Normalization layers	627
16.2.6	Dropout layers	627
16.2.7	Attention layers	628
16.2.8	Recurrent layers	631
16.2.9	Multiplicative layers	631
16.2.10	Implicit layers	632

16.3	Canonical examples of neural networks	632
16.3.1	Multilayer perceptrons (MLPs)	633
16.3.2	Convolutional neural networks (CNNs)	634
16.3.3	Autoencoders	635
16.3.4	Recurrent neural networks (RNNs)	636
16.3.5	Transformers	636
16.3.6	Graph neural networks (GNNs)	637
17	Bayesian neural networks	639
17.1	Introduction	639
17.2	Priors for BNNs	639
17.2.1	Gaussian priors	640
17.2.2	Sparsity-promoting priors	642
17.2.3	Learning the prior	642
17.2.4	Priors in function space	642
17.2.5	Architectural priors	643
17.3	Posterior for BNNs	643
17.3.1	Monte Carlo dropout	643
17.3.2	Laplace approximation	644
17.3.3	Variational inference	645
17.3.4	Expectation propagation	646
17.3.5	Last layer methods	646
17.3.6	SNGP	647
17.3.7	MCMC methods	647
17.3.8	Methods based on the SGD trajectory	648
17.3.9	Deep ensembles	649
17.3.10	Approximating the posterior predictive distribution	653
17.3.11	Tempered and cold posteriors	656
17.4	Generalization in Bayesian deep learning	657
17.4.1	Sharp vs flat minima	657
17.4.2	Mode connectivity and the loss landscape	658
17.4.3	Effective dimensionality of a model	659
17.4.4	The hypothesis space of DNNs	660
17.4.5	PAC-Bayes	661
17.4.6	Out-of-distribution generalization for BNNs	662
17.4.7	Model selection for BNNs	663
17.5	Online inference	664
17.5.1	Sequential Laplace for DNNs	664
17.5.2	Extended Kalman filtering for DNNs	665
17.5.3	Assumed density filtering for DNNs	667
17.5.4	Online variational inference for DNNs	668
17.6	Hierarchical Bayesian neural networks	669
17.6.1	Example: multimoons classification	670
18	Gaussian processes	673
18.1	Introduction	673
18.1.1	GPs: what and why?	673
18.2	Mercer kernels	675
18.2.1	Stationary kernels	676
18.2.2	Nonstationary kernels	681
18.2.3	Kernels for nonvectorial (structured) inputs	682
18.2.4	Making new kernels from old	682
18.2.5	Mercer's theorem	683
18.2.6	Approximating kernels with random features	684
18.3	GPs with Gaussian likelihoods	685
18.3.1	Predictions using noise-free observations	685
18.3.2	Predictions using noisy observations	686

18.3.3	Weight space vs function space	687
18.3.4	Semiparametric GPs	688
18.3.5	Marginal likelihood	689
18.3.6	Computational and numerical issues	689
18.3.7	Kernel ridge regression	690
18.4	GPs with non-Gaussian likelihoods	693
18.4.1	Binary classification	694
18.4.2	Multiclass classification	695
18.4.3	GPs for Poisson regression (Cox process)	696
18.4.4	Other likelihoods	696
18.5	Scaling GP inference to large datasets	697
18.5.1	Subset of data	697
18.5.2	Nyström approximation	698
18.5.3	Inducing point methods	699
18.5.4	Sparse variational methods	702
18.5.5	Exploiting parallelization and structure via kernel matrix multiplies	706
18.5.6	Converting a GP to an SSM	708
18.6	Learning the kernel	709
18.6.1	Empirical Bayes for the kernel parameters	709
18.6.2	Bayesian inference for the kernel parameters	712
18.6.3	Multiple kernel learning for additive kernels	713
18.6.4	Automatic search for compositional kernels	714
18.6.5	Spectral mixture kernel learning	717
18.6.6	Deep kernel learning	718
18.7	GPs and DNNs	720
18.7.1	Kernels derived from infinitely wide DNNs (NN-GP)	721
18.7.2	Neural tangent kernel (NTK)	723
18.7.3	Deep GPs	723
18.8	Gaussian processes for time series forecasting	724
18.8.1	Example: Mauna Loa	724
19	Beyond the iid assumption	727
19.1	Introduction	727
19.2	Distribution shift	727
19.2.1	Motivating examples	727
19.2.2	A causal view of distribution shift	729
19.2.3	The four main types of distribution shift	730
19.2.4	Selection bias	732
19.3	Detecting distribution shifts	732
19.3.1	Detecting shifts using two-sample testing	733
19.3.2	Detecting single out-of-distribution (OOD) inputs	733
19.3.3	Selective prediction	736
19.3.4	Open set and open world recognition	737
19.4	Robustness to distribution shifts	737
19.4.1	Data augmentation	738
19.4.2	Distributionally robust optimization	738
19.5	Adapting to distribution shifts	738
19.5.1	Supervised adaptation using transfer learning	738
19.5.2	Weighted ERM for covariate shift	740
19.5.3	Unsupervised domain adaptation for covariate shift	741
19.5.4	Unsupervised techniques for label shift	742
19.5.5	Test-time adaptation	742
19.6	Learning from multiple distributions	743
19.6.1	Multitask learning	743
19.6.2	Domain generalization	744
19.6.3	Invariant risk minimization	746

19.6.4	Meta learning	747
19.7	Continual learning	750
19.7.1	Domain drift	750
19.7.2	Concept drift	751
19.7.3	Class incremental learning	752
19.7.4	Catastrophic forgetting	754
19.7.5	Online learning	755
19.8	Adversarial examples	757
19.8.1	Whitebox (gradient-based) attacks	758
19.8.2	Blackbox (gradient-free) attacks	759
19.8.3	Real world adversarial attacks	760
19.8.4	Defenses based on robust optimization	761
19.8.5	Why models have adversarial examples	761

IV Generation 763

20 Generative models: an overview	765	
20.1	Introduction	765
20.2	Types of generative model	765
20.3	Goals of generative modeling	767
20.3.1	Generating data	767
20.3.2	Density estimation	769
20.3.3	Imputation	770
20.3.4	Structure discovery	771
20.3.5	Latent space interpolation	771
20.3.6	Latent space arithmetic	773
20.3.7	Generative design	774
20.3.8	Model-based reinforcement learning	774
20.3.9	Representation learning	774
20.3.10	Data compression	774
20.4	Evaluating generative models	774
20.4.1	Likelihood-based evaluation	775
20.4.2	Distances and divergences in feature space	776
20.4.3	Precision and recall metrics	777
20.4.4	Statistical tests	778
20.4.5	Challenges with using pretrained classifiers	779
20.4.6	Using model samples to train classifiers	779
20.4.7	Assessing overfitting	779
20.4.8	Human evaluation	780
21 Variational autoencoders	781	
21.1	Introduction	781
21.2	VAE basics	781
21.2.1	Modeling assumptions	782
21.2.2	Model fitting	783
21.2.3	Comparison of VAEs and autoencoders	783
21.2.4	VAEs optimize in an augmented space	784
21.3	VAE generalizations	786
21.3.1	β -VAE	787
21.3.2	InfoVAE	789
21.3.3	Multimodal VAEs	790
21.3.4	Semisupervised VAEs	793
21.3.5	VAEs with sequential encoders/decoders	794
21.4	Avoiding posterior collapse	796
21.4.1	KL annealing	797

21.4.2	Lower bounding the rate	798
21.4.3	Free bits	798
21.4.4	Adding skip connections	798
21.4.5	Improved variational inference	798
21.4.6	Alternative objectives	799
21.5	VAEs with hierarchical structure	799
21.5.1	Bottom-up vs top-down inference	800
21.5.2	Example: very deep VAE	801
21.5.3	Connection with autoregressive models	802
21.5.4	Variational pruning	804
21.5.5	Other optimization difficulties	804
21.6	Vector quantization VAE	805
21.6.1	Autoencoder with binary code	805
21.6.2	VQ-VAE model	805
21.6.3	Learning the prior	807
21.6.4	Hierarchical extension (VQ-VAE-2)	807
21.6.5	Discrete VAE	808
21.6.6	VQ-GAN	809
22	Autoregressive models	811
22.1	Introduction	811
22.2	Neural autoregressive density estimators (NADE)	812
22.3	Causal CNNs	812
22.3.1	1d causal CNN (convolutional Markov models)	813
22.3.2	2d causal CNN (PixelCNN)	813
22.4	Transformers	814
22.4.1	Text generation (GPT, etc.)	815
22.4.2	Image generation (DALL-E, etc.)	816
22.4.3	Other applications	818
23	Normalizing flows	819
23.1	Introduction	819
23.1.1	Preliminaries	819
23.1.2	How to train a flow model	821
23.2	Constructing flows	822
23.2.1	Affine flows	822
23.2.2	Elementwise flows	822
23.2.3	Coupling flows	825
23.2.4	Autoregressive flows	826
23.2.5	Residual flows	832
23.2.6	Continuous-time flows	834
23.3	Applications	836
23.3.1	Density estimation	836
23.3.2	Generative modeling	836
23.3.3	Inference	837
24	Energy-based models	839
24.1	Introduction	839
24.1.1	Example: products of experts (PoE)	840
24.1.2	Computational difficulties	840
24.2	Maximum likelihood training	841
24.2.1	Gradient-based MCMC methods	842
24.2.2	Contrastive divergence	842
24.3	Score matching (SM)	846
24.3.1	Basic score matching	846
24.3.2	Denoising score matching (DSM)	847
24.3.3	Sliced score matching (SSM)	848

24.3.4	Connection to contrastive divergence	849
24.3.5	Score-based generative models	850
24.4	Noise contrastive estimation	850
24.4.1	Connection to score matching	852
24.5	Other methods	853
24.5.1	Minimizing Differences/Derivatives of KL Divergences	853
24.5.2	Minimizing the Stein discrepancy	853
24.5.3	Adversarial training	854
25 Diffusion models	857	
25.1	Introduction	857
25.2	Denoising diffusion probabilistic models (DDPMs)	857
25.2.1	Encoder (forwards diffusion)	858
25.2.2	Decoder (reverse diffusion)	859
25.2.3	Model fitting	860
25.2.4	Learning the noise schedule	862
25.2.5	Example: image generation	863
25.3	Score-based generative models (SGMs)	864
25.3.1	Example	864
25.3.2	Adding noise at multiple scales	864
25.3.3	Equivalence to DDPM	866
25.4	Continuous time models using differential equations	867
25.4.1	Forwards diffusion SDE	867
25.4.2	Forwards diffusion ODE	868
25.4.3	Reverse diffusion SDE	869
25.4.4	Reverse diffusion ODE	870
25.4.5	Comparison of the SDE and ODE approach	871
25.4.6	Example	871
25.5	Speeding up diffusion models	871
25.5.1	DDIM sampler	872
25.5.2	Non-Gaussian decoder networks	872
25.5.3	Distillation	873
25.5.4	Latent space diffusion	874
25.6	Conditional generation	875
25.6.1	Conditional diffusion model	875
25.6.2	Classifier guidance	875
25.6.3	Classifier-free guidance	876
25.6.4	Generating high resolution images	876
25.7	Diffusion for discrete state spaces	877
25.7.1	Discrete Denoising Diffusion Probabilistic Models	877
25.7.2	Choice of Markov transition matrices for the forward processes	878
25.7.3	Parameterization of the reverse process	879
25.7.4	Noise schedules	880
25.7.5	Connections to other probabilistic models for discrete sequences	880
26 Generative adversarial networks	883	
26.1	Introduction	883
26.2	Learning by comparison	884
26.2.1	Guiding principles	885
26.2.2	Density ratio estimation using binary classifiers	886
26.2.3	Bounds on f -divergences	888
26.2.4	Integral probability metrics	890
26.2.5	Moment matching	892
26.2.6	On density ratios and differences	892
26.3	Generative adversarial networks	894
26.3.1	From learning principles to loss functions	894
26.3.2	Gradient descent	895

26.3.3	Challenges with GAN training	897
26.3.4	Improving GAN optimization	898
26.3.5	Convergence of GAN training	898
26.4	Conditional GANs	902
26.5	Inference with GANs	903
26.6	Neural architectures in GANs	904
26.6.1	The importance of discriminator architectures	904
26.6.2	Architectural inductive biases	905
26.6.3	Attention in GANs	905
26.6.4	Progressive generation	906
26.6.5	Regularization	907
26.6.6	Scaling up GAN models	908
26.7	Applications	908
26.7.1	GANs for image generation	908
26.7.2	Video generation	911
26.7.3	Audio generation	912
26.7.4	Text generation	912
26.7.5	Imitation learning	913
26.7.6	Domain adaptation	914
26.7.7	Design, art and creativity	914

V Discovery 915

27	Discovery methods: an overview	917
27.1	Introduction	917
27.2	Overview of Part V	918
28	Latent factor models	919
28.1	Introduction	919
28.2	Mixture models	919
28.2.1	Gaussian mixture models (GMMs)	920
28.2.2	Bernoulli mixture models	922
28.2.3	Gaussian scale mixtures (GSMs)	922
28.2.4	Using GMMs as a prior for inverse imaging problems	924
28.2.5	Using mixture models for classification problems	927
28.2.6	Unidentifiability	929
28.3	Factor analysis	930
28.3.1	Factor analysis: the basics	930
28.3.2	Probabilistic PCA	934
28.3.3	Mixture of factor analyzers	936
28.3.4	Factor analysis models for paired data	943
28.3.5	Factor analysis with exponential family likelihoods	946
28.3.6	Factor analysis with DNN likelihoods (VAEs)	947
28.3.7	Factor analysis with GP likelihoods (GP-LVM)	948
28.4	LFMs with non-Gaussian priors	949
28.4.1	Non-negative matrix factorization (NMF)	950
28.4.2	Multinomial PCA	952
28.5	Topic models	953
28.5.1	Latent Dirichlet allocation (LDA)	953
28.5.2	Correlated topic model	957
28.5.3	Dynamic topic model	957
28.5.4	LDA-HMM	958
28.6	Independent components analysis (ICA)	961
28.6.1	Noiseless ICA model	962
28.6.2	The need for non-Gaussian priors	963

28.6.3	Maximum likelihood estimation	965
28.6.4	Alternatives to MLE	965
28.6.5	Sparse coding	967
28.6.6	Nonlinear ICA	968
29 State-space models	969	
29.1	Introduction	969
29.2	Hidden Markov models (HMMs)	970
29.2.1	Conditional independence properties	970
29.2.2	State transition model	970
29.2.3	Discrete likelihoods	971
29.2.4	Gaussian likelihoods	972
29.2.5	Autoregressive likelihoods	972
29.2.6	Neural network likelihoods	973
29.3	HMMs: applications	974
29.3.1	Time series segmentation	974
29.3.2	Protein sequence alignment	976
29.3.3	Spelling correction	978
29.4	HMMs: parameter learning	980
29.4.1	The Baum-Welch (EM) algorithm	980
29.4.2	Parameter estimation using SGD	983
29.4.3	Parameter estimation using spectral methods	984
29.4.4	Bayesian HMMs	985
29.5	HMMs: generalizations	987
29.5.1	Hidden semi-Markov model (HSMM)	987
29.5.2	Hierarchical HMMs	989
29.5.3	Factorial HMMs	991
29.5.4	Coupled HMMs	992
29.5.5	Dynamic Bayes nets (DBN)	993
29.5.6	Changepoint detection	993
29.6	Linear dynamical systems (LDSs)	996
29.6.1	Conditional independence properties	996
29.6.2	Parameterization	996
29.7	LDS: applications	997
29.7.1	Object tracking and state estimation	997
29.7.2	Online Bayesian linear regression (recursive least squares)	998
29.7.3	Adaptive filtering	1000
29.7.4	Time series forecasting	1000
29.8	LDS: parameter learning	1001
29.8.1	EM for LDS	1001
29.8.2	Subspace identification methods	1003
29.8.3	Ensuring stability of the dynamical system	1003
29.8.4	Bayesian LDS	1004
29.8.5	Online parameter learning for SSMs	1005
29.9	Switching linear dynamical systems (SLDSs)	1005
29.9.1	Parameterization	1005
29.9.2	Posterior inference	1005
29.9.3	Application: Multitarget tracking	1006
29.10	Nonlinear SSMs	1010
29.10.1	Example: object tracking and state estimation	1010
29.10.2	Posterior inference	1011
29.11	Non-Gaussian SSMs	1011
29.11.1	Example: spike train modeling	1011
29.11.2	Example: stochastic volatility models	1012
29.11.3	Posterior inference	1013
29.12	Structural time series models	1013

29.12.1	Introduction	1013
29.12.2	Structural building blocks	1014
29.12.3	Model fitting	1016
29.12.4	Forecasting	1017
29.12.5	Examples	1017
29.12.6	Causal impact of a time series intervention	1021
29.12.7	Prophet	1025
29.12.8	Neural forecasting methods	1025
29.13	Deep SSMs	1026
29.13.1	Deep Markov models	1027
29.13.2	Recurrent SSM	1028
29.13.3	Improving multistep predictions	1029
29.13.4	Variational RNNs	1030
30	Graph learning	1033
30.1	Introduction	1033
30.2	Latent variable models for graphs	1033
30.3	Graphical model structure learning	1033
30.3.1	Methods	1033
30.3.2	Applications	1034
31	Nonparametric Bayesian models	1037
31.1	Introduction	1037
32	Representation learning	1039
32.1	Introduction	1039
32.2	Evaluating and comparing learned representations	1039
32.2.1	Downstream performance	1040
32.2.2	Representational similarity	1042
32.3	Approaches for learning representations	1046
32.3.1	Supervised representation learning and transfer	1047
32.3.2	Generative representation learning	1049
32.3.3	Self-supervised representation learning	1051
32.3.4	Multiview representation learning	1054
32.4	Theory of representation learning	1059
32.4.1	Identifiability	1059
32.4.2	Information maximization	1060
33	Interpretability	1063
33.1	Introduction	1063
33.1.1	The role of interpretability: unknowns and under-specifications	1064
33.1.2	Terminology and framework	1065
33.2	Methods for interpretable machine learning	1068
33.2.1	Inherently interpretable models: the model is its explanation	1069
33.2.2	Semi-inherently interpretable models: example-based methods	1071
33.2.3	Post-hoc or joint training: the explanation gives a partial view of the model	1071
33.2.4	Transparency and visualization	1075
33.3	Properties: the abstraction between context and method	1076
33.3.1	Properties of explanations from interpretable machine learning	1076
33.3.2	Properties of explanations from cognitive science	1078
33.4	Evaluation of interpretable machine learning models	1079
33.4.1	Computational evaluation: does the method have desired properties?	1080
33.4.2	User study-based evaluation: does the method help a user perform a target task?	1084
33.5	Discussion: how to think about interpretable machine learning	1088

VI Action 1093

34 Decision making under uncertainty	1095
34.1 Statistical decision theory	1095
34.1.1 Basics	1095
34.1.2 Frequentist decision theory	1095
34.1.3 Bayesian decision theory	1096
34.1.4 Frequentist optimality of the Bayesian approach	1097
34.1.5 Examples of one-shot decision making problems	1097
34.2 Decision (influence) diagrams	1101
34.2.1 Example: oil wildcatter	1102
34.2.2 Information arcs	1103
34.2.3 Value of information	1103
34.2.4 Computing the optimal policy	1104
34.3 A/B testing	1105
34.3.1 A Bayesian approach	1105
34.3.2 Example	1108
34.4 Contextual bandits	1109
34.4.1 Types of bandit	1110
34.4.2 Applications	1111
34.4.3 Exploration-exploitation tradeoff	1111
34.4.4 The optimal solution	1112
34.4.5 Upper confidence bounds (UCBs)	1113
34.4.6 Thompson sampling	1115
34.4.7 Regret	1116
34.5 Markov decision problems	1118
34.5.1 Basics	1118
34.5.2 Partially observed MDPs	1119
34.5.3 Episodes and returns	1119
34.5.4 Value functions	1121
34.5.5 Optimal value functions and policies	1121
34.6 Planning in an MDP	1122
34.6.1 Value iteration	1123
34.6.2 Policy iteration	1124
34.6.3 Linear programming	1125
34.7 Active learning	1126
34.7.1 Active learning scenarios	1126
34.7.2 Relationship to other forms of sequential decision making	1127
34.7.3 Acquisition strategies	1128
34.7.4 Batch active learning	1130
35 Reinforcement learning	1135
35.1 Introduction	1135
35.1.1 Overview of methods	1135
35.1.2 Value-based methods	1137
35.1.3 Policy search methods	1137
35.1.4 Model-based RL	1137
35.1.5 Exploration-exploitation tradeoff	1138
35.2 Value-based RL	1140
35.2.1 Monte Carlo RL	1140
35.2.2 Temporal difference (TD) learning	1140
35.2.3 TD learning with eligibility traces	1141
35.2.4 SARSA: on-policy TD control	1142
35.2.5 Q-learning: off-policy TD control	1143
35.2.6 Deep Q-network (DQN)	1144
35.3 Policy-based RL	1146
35.3.1 The policy gradient theorem	1147

35.3.2	REINFORCE	1148
35.3.3	Actor-critic methods	1148
35.3.4	Bound optimization methods	1150
35.3.5	Deterministic policy gradient methods	1152
35.3.6	Gradient-free methods	1153
35.4	Model-based RL	1153
35.4.1	Model predictive control (MPC)	1153
35.4.2	Combining model-based and model-free	1155
35.4.3	MBRL using Gaussian processes	1156
35.4.4	MBRL using DNNs	1157
35.4.5	MBRL using latent-variable models	1158
35.4.6	Robustness to model errors	1160
35.5	Off-policy learning	1160
35.5.1	Basic techniques	1161
35.5.2	The curse of horizon	1164
35.5.3	The deadly triad	1165
35.6	Control as inference	1167
35.6.1	Maximum entropy reinforcement learning	1167
35.6.2	Other approaches	1169
35.6.3	Imitation learning	1170
36 Causality		1173
36.1	Introduction	1173
36.2	Causal formalism	1175
36.2.1	Structural causal models	1175
36.2.2	Causal DAGs	1176
36.2.3	Identification	1178
36.2.4	Counterfactuals and the causal hierarchy	1180
36.3	Randomized control trials	1182
36.4	Confounder adjustment	1183
36.4.1	Causal estimand, statistical estimand, and identification	1183
36.4.2	ATE estimation with observed confounders	1186
36.4.3	Uncertainty quantification	1191
36.4.4	Matching	1191
36.4.5	Practical considerations and procedures	1192
36.4.6	Summary and practical advice	1195
36.5	Instrumental variable strategies	1197
36.5.1	Additive unobserved confounding	1198
36.5.2	Instrument monotonicity and local average treatment effect	1200
36.5.3	Two stage least squares	1203
36.6	Difference in differences	1204
36.6.1	Estimation	1207
36.7	Credibility checks	1208
36.7.1	Placebo checks	1208
36.7.2	Sensitivity analysis to unobserved confounding	1209
36.8	The do-calculus	1217
36.8.1	The three rules	1217
36.8.2	Revisiting backdoor adjustment	1218
36.8.3	Frontdoor adjustment	1219
36.9	Further reading	1220
Index		1223
Bibliography		1241

Preface

I am writing a longer [book] than usual because there is not enough time to write a short one.
(Blaise Pascal, paraphrased.)

This book is a sequel to [Mur22], and provides a deeper dive into various topics in machine learning (ML). The previous book mostly focused on techniques for learning functions of the form $f : \mathcal{X} \rightarrow \mathcal{Y}$, where f is some nonlinear model, such as a deep neural network, \mathcal{X} is the set of possible inputs (typically $\mathcal{X} = \mathbb{R}^D$), and $\mathcal{Y} = \{1, \dots, C\}$ represents the set of labels for classification problems or $\mathcal{Y} = \mathbb{R}$ for regression problems. Judea Pearl, a well known AI researcher, has called this kind of ML a form of “glorified curve fitting” (quoted in [Har18]).

In this book, we expand the scope of ML to encompass more challenging problems. For example, we consider training and testing under different distributions; we consider generation of high dimensional outputs, such as images, text, and graphs, so the output space is, say, $\mathcal{Y} = \mathbb{R}^{256 \times 256}$; we discuss methods for discovering “insights” about data, based on latent variable models; and we discuss how to use probabilistic models for causal inference and decision making under uncertainty.

We assume the reader has some prior exposure to ML and other relevant mathematical topics (e.g., probability, statistics, linear algebra, optimization). This background material can be found in the prequel to this book, [Mur22], as well several other good books (e.g., [Lin+21b; DFO20]).

Python code (mostly in JAX) to reproduce nearly all of the figures can be found online. In particular, if a figure caption says “Generated by `gauss_plot_2d.ipynb`”, then you can find the corresponding Jupyter notebook at probml.github.io/notebooks#gauss_plot_2d.ipynb. Clicking on the figure link in the pdf version of the book will take you to this list of notebooks. Clicking on the notebook link will open it inside Google Colab, which will let you easily reproduce the figure for yourself, and modify the underlying source code to gain a deeper understanding of the methods. (Colab gives you access to a free GPU, which is useful for some of the more computationally heavy demos.)

In addition to the online code, probml.github.io/supp contains some additional supplementary online content which was excluded from the main book for space reasons. For exercises (and solutions) related to the topics in this book, see [Gut22].

Contributing authors

This book is the result of a lot of effort from a lot of people. I would especially like to thank the following people who wrote or cowrote various sections or chapters:

- Alex Alemi (Google), who co-wrote [Section 5.1 \(KL divergence\)](#) (with Murphy).
- Jeff Bilmes (U. Washington), who wrote [Section 6.9 \(Submodular optimization\)](#).
- Peter Chang, who co-wrote [Section 8.5.1 \(General Gaussian filtering\)](#) (with Murphy).
- Marco Cuturi (Apple, work done at Google), who wrote [Section 6.8 \(Optimal transport\)](#).
- Alexander D’Amour (Google), who co-wrote [Chapter 36 \(Causality\)](#) (with Veitch).
- Finale Doshi-Velez (Harvard), who co-wrote [Chapter 33 \(Interpretability\)](#) (with Kim).
- Roy Frostig (Google), who wrote [Section 6.2 \(Automatic differentiation\)](#).
- Justin Gilmer (Google), who wrote [Section 19.8 \(Adversarial examples\)](#).
- Giles Harper-Donnelly, who wrote [Section 8.2.4 \(Information form filtering and smoothing\)](#).
- Been Kim (Google), who co-wrote [Chapter 33 \(Interpretability\)](#) (with Doshi-Velez).
- Durk Kingma (Google), who co-wrote [Chapter 24 \(Energy-based models\)](#) (with Song).
- Simon Kornblith (Google), who co-wrote [Chapter 32 \(Representation learning\)](#) (with Poole).
- Balaji Lakshminarayanan (Google), who co-wrote [Chapter 23 \(Normalizing flows\)](#) (with Papamakarios) and [Chapter 26 \(Generative adversarial networks\)](#) (with Mohamed and Rosca).
- Lihong Li (Amazon, work done at Google), who co-wrote [Section 34.4 \(Contextual bandits\)](#) and [Chapter 35 \(Reinforcement learning\)](#) (with Murphy).
- Xinglong Li (UBC), who wrote [Section 15.2.9 \(Multivariate linear regression\)](#), [Section 29.4.4.1 \(Blocked Gibbs sampling for HMMs\)](#), [Section 29.8.4.1 \(Blocked Gibbs sampling for LDS\)](#), and [Supplementary Section 31.2.3](#).
- Shakir Mohamed (Deepmind), who co-wrote [Chapter 26 \(Generative adversarial networks\)](#) (with Lakshminarayanan and Rosca).
- George Papamakarios (Deepmind), who cowrote [Chapter 23 \(Normalizing flows\)](#) (with Lakshminarayanan).
- Zeel Patel (IIT Gandhinagar), who cowrote [Section 34.7 \(Active learning\)](#) (with Murphy).
- Ben Poole (Google), who co-wrote [Chapter 32 \(Representation learning\)](#) (with Kornblith).
- Mihaela Rosca (Deepmind/UCL), who co-wrote [Chapter 26 \(Generative adversarial networks\)](#).
- Vinayak Rao (Purdue), who wrote [Chapter 31 \(Nonparametric Bayesian models\)](#).
- Yang Song (Stanford), who co-wrote [Chapter 24 \(Energy-based models\)](#) (with Kingma).
- Victor Veitch (Google/U. Chicago), who co-wrote [Chapter 36 \(Causality\)](#) (with D’Amour).
- Andrew Wilson (NYU), who co-wrote [Chapter 17 \(Bayesian neural networks\)](#) and [Chapter 18 \(Gaussian processes\)](#) (with Murphy).

Other contributors

I would also like to thank the following people who helped in various other ways:

- Many people who helped make or improve the figures, including: Aman Atman, Vibhuti Bansal, Shobhit Belwal, Aadesh Desai, Vishal Ghoniya, Anand Hegde, Ankita Kumari Jain, Madhav Kanda, Aleyna Kara, Rohit Khoiwal, Taksh Panchal, Dhruv Patel, Prey Patel, Nitish Sharma, Hetvi Shastri, Mahmoud Soliman, and Gautam Vashishtha. A special shout out to Zeel B Patel and Karm Patel for their significant efforts in improving the figure quality.
- Participants in the Google Summer of Code (GSOC) for 2021, including Ming Liang Ang, Aleyna Kara, Gerardo Duran-Martin, Srikanth Reddy Jilugu, Drishti Patel, and co-mentor Mahmoud Soliman.
- Participants in the Google Summer of Code (GSOC) for 2022, including Peter Chang, Giles

Harper-Donnelly, Xinglong Li, Zeel B Patel, Karm Patel, Qingyao Sun, and co-mentors Nipun Batra and Scott Linderman.

- Many other people who contributed code (see autogenerated list at <https://github.com/probml/pyprobml#acknowledgements>).
- Many people who proofread parts of the book, including: Aalto Seminar students, Bill Behrman, Kay Brodersen, Peter Chang, Krzysztof Choromanski, Adrien Corenflos, Tom Dietterich, Gerardo Duran-Martin, Lehman Krunoslav, Ruiqi Gao, Amir Globerson, Giles Harper-Donnelly, Ravin Kumar, Junpeng Lao, Stephen Mandt, Norm Matloff, Simon Prince, Rif Saurous, Erik Sudderth, Donna Vakalis, Hal Varian, Chris Williams, Raymond Yeh, and others listed at <https://github.com/probml/pml2-book/issues?q=is%3Aissue>. A special shout out to John Fearn who proofread almost all the math, and the MIT Press editor who ensured I use “Oxford commas” in all the right places.

About the cover

The cover illustrates a variational autoencoder (Chapter 21) being used to map from a 2d Gaussian to image space.

Kevin Patrick Murphy
Palo Alto, California
August 2023.

Changelog: see <https://github.com/probml/pml2-book/issues?q=is%3Aissue+is%3Aclosed>.

1 Introduction

“Intelligence is not just about pattern recognition and function approximation. It’s about modeling the world”. — Josh Tenenbaum, NeurIPS 2021.

Much of current machine learning focuses on the task of mapping inputs to outputs (i.e., approximating functions of the form $f : \mathcal{X} \rightarrow \mathcal{Y}$), often using “**deep learning**” (see e.g., [LBH15; Sch14; Sej20; BLH21]). Judea Pearl, a well known AI researcher, has called this “glorified curve fitting” (quoted in [Har18]). This is a little unfair, since when \mathcal{X} and/or \mathcal{Y} are high-dimensional spaces — such as images, sentences, graphs, or sequences of decisions/actions — then the term “curve fitting” is rather misleading, since one-dimensional intuitions often do not work in higher-dimensional settings (see e.g., [BPL21a]). Nevertheless, the quote gets at what many feel is lacking in current attempts to “solve AI” using machine learning techniques, namely that they are too focused on prediction of observable patterns, and not focused enough on “understanding” the underlying *latent structure* behind these patterns.

Gaining a “deep understanding” of the structure behind the observed data is necessary for advancing science, as well as for certain applications, such as healthcare (see e.g., [DD22]), where identifying the *root causes* or mechanisms behind various diseases is the key to developing cures. In addition, such “deep understanding” is necessary in order to develop *robust* and *efficient* systems. By “robust” we mean methods that work well even if there are unexpected changes to the data distribution to which the system is applied, which is an important concern in many areas, such as robotics (see e.g., [Roy+21]). By “efficient” we generally mean data or statistically efficient, i.e., methods that can learn quickly from small amounts of data (cf., [Lu+21b]). This is important since data can be limited in some domains, such as healthcare and robotics, even though it is abundant in other domains, such as language and vision, due to the ability to scrape the internet. We are also interested in computationally efficient methods, although this is a secondary concern as computing power continues to grow. (We also note that this trend has been instrumental to much of the recent progress in AI, as noted in [Sut19].)

To develop robust and efficient systems, this book adopts a model-based approach, in which we try to learn *parsimonious representations* of the underlying “**data generating process**” (**DGP**) given samples from one or more datasets (c.f., [Lak+17; Win+19; Sch20; Ben+21a; Cun22; MTS22]). This is in fact similar to the scientific method, where we try to explain (features of) the observations by developing theories or models. One way to formalize this process is in terms of **Bayesian inference** applied to probabilistic models, as argued in [Jay03; Box80; GS13]. We discuss inference algorithms in detail in Part II of the book.¹ But before we get there, in Part I we cover some relevant background

1. Note that, in the deep learning community, the term “inference” means applying a function to some inputs to

material that will be needed. (This part can be skipped by readers who are already familiar with these basics.)

Once we have a set of inference methods in our toolbox (some of which may be as simple as computing a maximum likelihood estimate using an optimization method, such as stochastic gradient descent) we can turn our focus to discussing different kinds of models. The choice of model depends on our task, the kind and amount of data we have, and our metric(s) of success. We will broadly consider four main kinds of task: prediction (e.g., classification and regression), generation (e.g., of images or text), discovery (of “meaningful structure” in data), and control (optimal decision making). We give more details below.

In Part III, we discuss models for prediction. These models are conditional distributions of the form $p(\mathbf{y}|\mathbf{x})$, where $\mathbf{x} \in \mathcal{X}$ is some input (often high dimensional), and $\mathbf{y} \in \mathcal{Y}$ is the desired output (often low dimensional). In this part of the book, we assume there is one right answer that we want to predict, although we may be uncertain about it.

In Part IV, we discuss models for generation. These models are distributions of the form $p(\mathbf{x})$ or $p(\mathbf{x}|\mathbf{c})$, where \mathbf{c} are optional conditioning inputs, and where there may be multiple valid outputs. For example, given a text prompt \mathbf{c} , we may want to generate a diverse set of images \mathbf{x} that “match” the caption. Evaluating such models is harder than in the prediction setting, since it is less clear what the desired output should be.

In Part V, we discuss latent variable models, which are joint models of the form $p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$, where \mathbf{z} is the hidden state and \mathbf{x} are the observations that are assumed to be generated from \mathbf{z} . The goal is to compute $p(\mathbf{z}|\mathbf{x})$, in order to uncover some (hopefully meaningful/useful) underlying state or patterns in the observed data. We also consider methods for trying to discover patterns learned implicitly by predictive models of the form $p(\mathbf{y}|\mathbf{x})$, without relying on an explicit generative model of the data.

Finally, in Part VI, we discuss models and algorithms which can be used to make decisions under uncertainty. This naturally leads into the very important topic of causality, with which we close the book.

In view of the broad scope of the book, we cannot go into detail on every topic. However, we have attempted to cover all the basics. In some cases, we also provide a “deeper dive” into the research frontier (as of 2022). We hope that by bringing all these topics together, you will find it easier to make connections between all these seemingly disparate areas, and can thereby deepen your understanding of the field of machine learning.

compute the output. This is unrelated to Bayesian inference, which is concerned with the much harder task of inverting a function, and working backwards from observed outputs to possible hidden inputs (causes). The latter is more closely related to what the deep learning community calls “training”.

PART I

Fundamentals

2 Probability

2.1 Introduction

In this section, we formally define what we mean by probability, following the presentation of [Cha21, Ch. 2]. Other good introductions to this topic can be found in e.g., [GS97; BT08; Bet18; DFO20].

2.1.1 Probability space

We define a **probability space** to be a triple $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the **sample space**, which is the set of possible outcomes from an experiment; \mathcal{F} is the **event space**, which is the set of all possible subsets of Ω ; and \mathbb{P} is the **probability measure**, which is a mapping from an event $E \subseteq \Omega$ to a number in $[0, 1]$ (i.e., $\mathbb{P} : \mathcal{F} \rightarrow [0, 1]$), which satisfies certain consistency requirements, which we discuss in Section 2.1.4.

2.1.2 Discrete random variables

The simplest setting is where the outcomes of the experiment constitute a countable set. For example, consider throwing a 3-sided die, where the faces are labeled “A”, “B”, and “C”. (We choose 3 sides instead of 6 for brevity.) The sample space is $\Omega = \{A, B, C\}$, which represents all the possible outcomes of the “experiment”. The event space is the set of all possible subsets of the sample space, so $\mathcal{F} = \{\emptyset, \{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$. An event is an element of the event space. For example, the event $E_1 = \{A, B\}$ represents outcomes where the die shows face A or B, and event $E_2 = \{C\}$ represents the outcome where the die shows face C.

Once we have defined the event space, we need to specify the probability measure, which provides a way to compute the “size” or “weight” of each set in the event space. In the 3-sided die example, suppose we define the probability of each outcome (atomic event) as $\mathbb{P}[\{A\}] = \frac{2}{6}$, $\mathbb{P}[\{B\}] = \frac{1}{6}$, and $\mathbb{P}[\{C\}] = \frac{3}{6}$. We can derive the probability of other events by adding up the measures for each outcome, e.g., $\mathbb{P}[\{A, B\}] = \frac{2}{6} + \frac{1}{6} = \frac{1}{2}$. We formalize this in Section 2.1.4.

To simplify notation, we will assign a number to each possible outcome in the event space. This can be done by defining a **random variable** or **rv**, which is a function $X : \Omega \rightarrow \mathbb{R}$ that maps an outcome $\omega \in \Omega$ to a number $X(\omega)$ on the real line. For example, we can define the random variable X for our 3-sided die using $X(A) = 1$, $X(B) = 2$, $X(C) = 3$. As another example, consider an experiment where we flip a fair coin twice. The sample space is $\Omega = \{\omega_1 = (H, H), \omega_2 = (H, T), \omega_3 = (T, H), \omega_4 = (T, T)\}$, where H stands for head, and T for tail. Let X be the random variable that represents the number of heads. Then we have $X(\omega_1) = 2$, $X(\omega_2) = 1$, $X(\omega_3) = 1$, and $X(\omega_4) = 0$.

We define the set of possible values of the random variable to be its **state space**, denoted $X(\Omega) = \mathcal{X}$. We define the probability of any given state using

$$p_X(a) = \mathbb{P}[X = a] = \mathbb{P}[X^{-1}(a)] \quad (2.1)$$

where $X^{-1}(a) = \{\omega \in \Omega | X(\omega) = a\}$ is the pre-image of a . Here p_X is called the **probability mass function** or **pmf** for random variable X . In the example where we flip a fair coin twice, the pmf is $p_X(0) = \mathbb{P}[\{(T, T)\}] = \frac{1}{4}$, $p_X(1) = \mathbb{P}[\{(T, H), (H, T)\}] = \frac{2}{4}$, and $p_X(2) = \mathbb{P}[\{(H, H)\}] = \frac{1}{4}$. The pmf can be represented by a histogram, or some parametric function (see Section 2.2.1). We call p_X the **probability distribution** for rv X . We will often drop the X subscript from p_X where it is clear from context.

2.1.3 Continuous random variables

We can also consider experiments with continuous outcomes. In this case, we assume the sample space is a subset of the reals, $\Omega \subseteq \mathbb{R}$, and we define each continuous random variable to be the identify function, $X(\omega) = \omega$.

For example, consider measuring the duration of some event (in seconds). We define the sample space to be $\Omega = \{t : 0 \leq t \leq T_{\max}\}$. Since this is an uncountable set, we cannot define all possible subsets by enumeration, unlike the discrete case. Instead, we need to define event space in terms of a **Borel sigma-field**, also called a **Borel sigma-algebra**. We say that \mathcal{F} is a σ -field if (1) $\emptyset \in \mathcal{F}$ and $\Omega \in \mathcal{F}$; (2) \mathcal{F} is closed under complement, so if $E \in \mathcal{F}$ then $E^c \in \mathcal{F}$; and (3) \mathcal{F} is closed under countable unions and intersections, meaning that $\cup_{i=1}^{\infty} E_i \in \mathcal{F}$ and $\cap_{i=1}^{\infty} E_i \in \mathcal{F}$, provided $E_1, E_2, \dots \in \mathcal{F}$. Finally, we say that \mathcal{B} is a Borel σ -field if it is a σ -field generated from semi-closed intervals of the form $(-\infty, b] = \{x : -\infty < x \leq b\}$. By taking unions, intersections and complements of these intervals, we can see that \mathcal{B} contains the following sets:

$$(a, b), [a, b], (a, b], [a, b], \{b\}, -\infty \leq a \leq b \leq \infty \quad (2.2)$$

In our duration example, we can further restrict the event space to only contain intervals whose lower bound is 0 and whose upper bound is $\leq T_{\max}$.

To define the probability measure, we assign a weighting function $p_X(x) \geq 0$ for each $x \in \Omega$ known as a **probability density function** or **pdf**. See Section 2.2.2 for a list of common pdf's. We can then derive the probability of an event $E = [a, b]$ using

$$\mathbb{P}([a, b]) = \int_E d\mathbb{P} = \int_a^b p(x) dx \quad (2.3)$$

We can also define the **cumulative distribution function** or **cdf** for random variable X as follows:

$$P_X(x) \triangleq \mathbb{P}[X \leq x] = \int_{-\infty}^x p_X(x') dx' \quad (2.4)$$

From this we can compute the probability of an interval using

$$\mathbb{P}([a, b]) = p(a \leq X \leq b) = P_X(b) - P_X(a) \quad (2.5)$$

The term “probability distribution” could refer to the pdf p_X or the cdf P_X or even the probability measure \mathbb{P} .

We can generalize the above definitions to multidimensional spaces, $\Omega \subseteq \mathbb{R}^n$, as well as more complex sample spaces, such as functions.

2.1.4 Probability axioms

The probability law associated with the event space must follow the **axioms of probability**, also called the **Kolmogorov axioms**, which are as follows:¹

- Non-negativity: $\mathbb{P}[E] \geq 0$ for any $E \subseteq \Omega$.
- Normalization: $\mathbb{P}[\Omega] = 1$.
- Additivity: for any countable sequence of pairwise disjoint sets $\{E_1, E_2, \dots\}$, we have

$$\mathbb{P}[\cup_{i=1}^{\infty} E_i] = \sum_{i=1}^{\infty} \mathbb{P}[E_i] \quad (2.6)$$

In the finite case, where we just have two disjoint sets, E_1 and E_2 , this becomes

$$\mathbb{P}[E_1 \cup E_2] = \mathbb{P}[E_1] + \mathbb{P}[E_2] \quad (2.7)$$

This corresponds to the probability of event E_1 or E_2 , assuming they are mutually exclusive (disjoint sets).

From these axioms, we can derive the **complement rule**:

$$\mathbb{P}[E^c] = 1 - \mathbb{P}[E] \quad (2.8)$$

where $E^c = \Omega \setminus E$ is the complement of E . (This follows since $\mathbb{P}[\Omega] = 1 = \mathbb{P}[E \cup E^c] = \mathbb{P}[E] + \mathbb{P}[E^c]$.) We can also show that $\mathbb{P}[E] \leq 1$ (proof by contradiction), and $\mathbb{P}[\emptyset] = 0$ (which follows from first corollary with $E = \Omega$).

We can also show the following result, known as the **addition rule**:

$$\mathbb{P}[E_1 \cup E_2] = \mathbb{P}[E_1] + \mathbb{P}[E_2] - \mathbb{P}[E_1 \cap E_2] \quad (2.9)$$

This holds for any pair of events, even if they are not disjoint.

2.1.5 Conditional probability

Consider two events E_1 and E_2 . If $\mathbb{P}[E_2] \neq 0$, we define the **conditional probability** of E_1 given E_2 as

$$\mathbb{P}[E_1|E_2] \triangleq \frac{\mathbb{P}[E_1 \cap E_2]}{\mathbb{P}[E_2]} \quad (2.10)$$

From this, we can get the **multiplication rule**:

$$\mathbb{P}[E_1 \cap E_2] = \mathbb{P}[E_1|E_2]\mathbb{P}[E_2] = \mathbb{P}[E_2|E_1]\mathbb{P}[E_1] \quad (2.11)$$

1. These laws can be shown to follow from a more basic set of assumptions about reasoning under uncertainty, a result known as **Cox's theorem** [Cox46; Cox61].

Conditional probability measures how likely an event E_1 is given that event E_2 has happened. However, if the events are unrelated, the probability will not change. Formally, we say that E_1 and E_2 are **independent events** if

$$\mathbb{P}[E_1 \cap E_2] = \mathbb{P}[E_1]\mathbb{P}[E_2] \quad (2.12)$$

If both $\mathbb{P}[E_1] > 0$ and $\mathbb{P}[E_2] > 0$, this is equivalent to requiring that $\mathbb{P}[E_1|E_2] = \mathbb{P}[E_1]$ or equivalently, $\mathbb{P}[E_2|E_1] = \mathbb{P}[E_2]$. Similarly, we say that E_1 and E_2 are conditionally independent given E_3 if

$$\mathbb{P}[E_1 \cap E_2|E_3] = \mathbb{P}[E_1|E_3]\mathbb{P}[E_2|E_3] \quad (2.13)$$

From the definition of conditional probability, we can derive the **law of total probability**, which states the following: if $\{A_1, \dots, A_n\}$ is a partition of the sample space Ω , then for any event $B \subseteq \Omega$, we have

$$\mathbb{P}[B] = \sum_{i=1}^n \mathbb{P}[B|A_i]\mathbb{P}[A_i] \quad (2.14)$$

2.1.6 Bayes' rule

From the definition of conditional probability, we can derive **Bayes' rule**, also called **Bayes' theorem**, which says that, for any two events E_1 and E_2 such that $\mathbb{P}[E_1] > 0$ and $\mathbb{P}[E_2] > 0$, we have

$$\mathbb{P}[E_1|E_2] = \frac{\mathbb{P}[E_2|E_1]\mathbb{P}[E_1]}{\mathbb{P}[E_2]} \quad (2.15)$$

For a discrete random variable X with K possible states, we can write Bayes' rule as follows, using the law of total probability:

$$p(X = k|E) = \frac{p(E|X = k)p(X = k)}{p(E)} = \frac{p(E|X = k)p(X = k)}{\sum_{k'=1}^K p(E|X = k')p(X = k')} \quad (2.16)$$

Here $p(X = k)$ is the **prior probability**, $p(E|X = k)$ is the **likelihood**, $p(X = k|E)$ is the **posterior probability**, and $p(E)$ is a normalization constant, known as the **marginal likelihood**.

Similarly, for a continuous random variable X , we can write Bayes' rule as follows:

$$p(X = x|E) = \frac{p(E|X = x)p(X = x)}{p(E)} = \frac{p(E|X = x)p(X = x)}{\int p(E|X = x')p(X = x')dx'} \quad (2.17)$$

2.2 Some common probability distributions

There are a wide variety of probability distributions that are used for various kinds of models. We summarize some of the more commonly used ones in the sections below. See [Supplementary Chapter 2](#) for more information, and <https://ben18785.shinyapps.io/distribution-zoo/> for an interactive visualization.

2.2.1 Discrete distributions

In this section, we discuss some discrete distributions defined on subsets of the (non-negative) integers.

2.2.1.1 Bernoulli and binomial distributions

Let $x \in \{0, 1, \dots, N\}$. The **binomial distribution** is defined by

$$\text{Bin}(x|N, \mu) \triangleq \binom{N}{x} \mu^x (1 - \mu)^{N-x} \quad (2.18)$$

where $\binom{N}{k} \triangleq \frac{N!}{(N-k)!k!}$ is the number of ways to choose k items from N (this is known as the **binomial coefficient**, and is pronounced “N choose k”).

If $N = 1$, so $x \in \{0, 1\}$, the binomial distribution reduces to the **Bernoulli distribution**:

$$\text{Ber}(x|\mu) = \begin{cases} 1 - \mu & \text{if } x = 0 \\ \mu & \text{if } x = 1 \end{cases} \quad (2.19)$$

where $\mu = \mathbb{E}[x] = p(x=1)$ is the mean.

2.2.1.2 Categorical and multinomial distributions

If the variable is discrete-valued, $x \in \{1, \dots, K\}$, we can use the **categorical distribution**:

$$\text{Cat}(x|\boldsymbol{\theta}) \triangleq \prod_{k=1}^K \theta_k^{\mathbb{I}(x=k)} \quad (2.20)$$

Alternatively, we can represent the K -valued variable x with the one-hot binary vector \mathbf{x} , which lets us write

$$\text{Cat}(\mathbf{x}|\boldsymbol{\theta}) \triangleq \prod_{k=1}^K \theta_k^{x_k} \quad (2.21)$$

If the k 'th element of \mathbf{x} counts the number of times the value k is seen in $N = \sum_{k=1}^K x_k$ trials, then we get the **multinomial distribution**:

$$\mathcal{M}(\mathbf{x}|N, \boldsymbol{\theta}) \triangleq \binom{N}{x_1 \dots x_K} \prod_{k=1}^K \theta_k^{x_k} \quad (2.22)$$

where the **multinomial coefficient** is defined as

$$\binom{N}{k_1 \dots k_m} \triangleq \frac{N!}{k_1! \dots k_m!} \quad (2.23)$$

2.2.1.3 Poisson distribution

Suppose $X \in \{0, 1, 2, \dots\}$. We say that a random variable has a **Poisson** distribution with parameter $\lambda > 0$, written $X \sim \text{Poi}(\lambda)$, if its pmf (probability mass function) is

$$\text{Poi}(x|\lambda) = e^{-\lambda} \frac{\lambda^x}{x!} \quad (2.24)$$

where λ is the mean (and variance) of x .

2.2.1.4 Negative binomial distribution

Suppose we have an “urn” with N balls, R of which are red and B of which are blue. Suppose we perform **sampling with replacement** until we get $n \geq 1$ balls. Let X be the number of these that are blue. It can be shown that $X \sim \text{Bin}(n, p)$, where $p = B/N$ is the fraction of blue balls; thus X follows the binomial distribution, discussed in Section 2.2.1.1.

Now suppose we consider drawing a red ball a “failure”, and drawing a blue ball a “success”. Suppose we keep drawing balls until we observe r failures. Let X be the resulting number of successes (blue balls); it can be shown that $X \sim \text{NegBinom}(r, p)$, which is the **negative binomial distribution** defined by

$$\text{NegBinom}(x|r, p) \triangleq \binom{x+r-1}{x} (1-p)^r p^x \quad (2.25)$$

for $x \in \{0, 1, 2, \dots\}$. (If r is real-valued, we replace $\binom{x+r-1}{x}$ with $\frac{\Gamma(x+r)}{x!\Gamma(r)}$, exploiting the fact that $(x-1)! = \Gamma(x)$.)

This distribution has the following moments:

$$\mathbb{E}[x] = \frac{p r}{1-p}, \quad \mathbb{V}[x] = \frac{p r}{(1-p)^2} \quad (2.26)$$

This two parameter family has more modeling flexibility than the Poisson distribution, since it can represent the mean and variance separately. This is useful, e.g., for modeling “contagious” events, which have positively correlated occurrences, causing a larger variance than if the occurrences were independent. In fact, the Poisson distribution is a special case of the negative binomial, since it can be shown that $\text{Poi}(\lambda) = \lim_{r \rightarrow \infty} \text{NegBinom}(r, \frac{\lambda}{1+\lambda})$. Another special case is when $r = 1$; this is called the **geometric distribution**.

2.2.2 Continuous distributions on \mathbb{R}

In this section, we discuss some univariate distributions defined on the reals, $p(x)$ for $x \in \mathbb{R}$.

2.2.2.1 Gaussian (Normal)

The most widely used univariate distribution is the **Gaussian distribution**, also called the **normal distribution**. (See [Mur22, Sec 2.6.4] for a discussion of these names.) The pdf (probability density function) of the Gaussian is given by

$$\mathcal{N}(x|\mu, \sigma^2) \triangleq \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \quad (2.27)$$

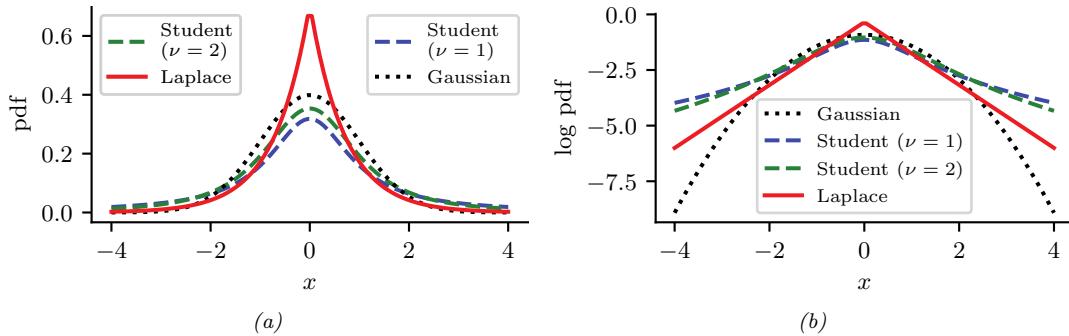


Figure 2.1: (a) The pdf's for a $\mathcal{N}(0, 1)$, $\mathcal{T}_1(0, 1)$ and Laplace($0, 1/\sqrt{2}$). The mean is 0 and the variance is 1 for both the Gaussian and Laplace. The mean and variance of the Student distribution is undefined when $\nu = 1$. (b) Log of these pdf's. Note that the Student distribution is not log-concave for any parameter value, unlike the Laplace distribution. Nevertheless, both are unimodal. Generated by [student_laplace_pdf_plot.ipynb](#).

where $\sqrt{2\pi\sigma^2}$ is the normalization constant needed to ensure the density integrates to 1. The parameter μ encodes the mean of the distribution, which is also equal to the mode. The parameter σ^2 encodes the variance. Sometimes we talk about the **precision** of a Gaussian, by which we mean the inverse variance: $\lambda = 1/\sigma^2$. A high precision means a narrow distribution (low variance) centered on μ .

The cumulative distribution function or cdf of the Gaussian is defined as

$$\Phi(x; \mu, \sigma^2) \triangleq \int_{-\infty}^x \mathcal{N}(z|\mu, \sigma^2) dz \quad (2.28)$$

If $\mu = 0$ and $\sigma = 1$ (known as the **standard normal** distribution), we just write $\Phi(x)$.

2.2.2.2 Half-normal

For some problems, we want a distribution over non-negative reals. One way to create such a distribution is to define $Y = |X|$, where $X \sim \mathcal{N}(0, \sigma^2)$. The induced distribution for Y is called the **half-normal distribution**, which has the pdf

$$\mathcal{N}_+(y|\sigma) \triangleq 2\mathcal{N}(y|0, \sigma^2) = \frac{\sqrt{2}}{\sigma\sqrt{\pi}} \exp\left(-\frac{y^2}{2\sigma^2}\right) \quad y \geq 0 \quad (2.29)$$

This can be thought of as the $\mathcal{N}(0, \sigma^2)$ distribution “folded over” onto itself.

2.2.2.3 Student *t*-distribution

One problem with the Gaussian distribution is that it is sensitive to outliers, since the probability decays exponentially fast with the (squared) distance from the center. A more robust distribution is the **Student *t*-distribution**, which we shall call the **Student distribution** for short. Its pdf is as

follows:

$$\mathcal{T}_\nu(x|\mu, \sigma^2) = \frac{1}{Z} \left[1 + \frac{1}{\nu} \left(\frac{x - \mu}{\sigma} \right)^2 \right]^{-\left(\frac{\nu+1}{2}\right)} \quad (2.30)$$

$$Z = \frac{\sqrt{\nu\pi\sigma^2}\Gamma(\frac{\nu}{2})}{\Gamma(\frac{\nu+1}{2})} = \sqrt{\nu}\sigma B\left(\frac{1}{2}, \frac{\nu}{2}\right) \quad (2.31)$$

where μ is the mean, $\sigma > 0$ is the scale parameter (not the standard deviation), and $\nu > 0$ is called the **degrees of freedom** (although a better term would be the **degree of normality** [Kru13], since large values of ν make the distribution act like a Gaussian). Here $\Gamma(a)$ is the **gamma function** defined by

$$\Gamma(a) \triangleq \int_0^\infty x^{a-1} e^{-x} dx \quad (2.32)$$

and $B(a, b)$ is the **beta function**, defined by

$$B(a, b) \triangleq \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \quad (2.33)$$

2.2.2.4 Cauchy distribution

If $\nu = 1$, the Student distribution is known as the **Cauchy** or **Lorentz** distribution. Its pdf is defined by

$$\mathcal{C}(x|\mu, \gamma) = \frac{1}{Z} \left[1 + \left(\frac{x - \mu}{\gamma} \right)^2 \right]^{-1} \quad (2.34)$$

where $Z = \gamma\beta(\frac{1}{2}, \frac{1}{2}) = \gamma\pi$. This distribution is notable for having such heavy tails that the integral that defines the mean does not converge.

The **half Cauchy** distribution is a version of the Cauchy (with mean 0) that is “folded over” on itself, so all its probability density is on the positive reals. Thus it has the form

$$\mathcal{C}_+(x|\gamma) \triangleq \frac{2}{\pi\gamma} \left[1 + \left(\frac{x}{\gamma} \right)^2 \right]^{-1} \quad (2.35)$$

2.2.2.5 Laplace distribution

Another distribution with heavy tails is the **Laplace distribution**, also known as the **double sided exponential** distribution. This has the following pdf:

$$\text{Laplace}(x|\mu, b) \triangleq \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right) \quad (2.36)$$

Here μ is a location parameter and $b > 0$ is a scale parameter. See Figure 2.1 for a plot.

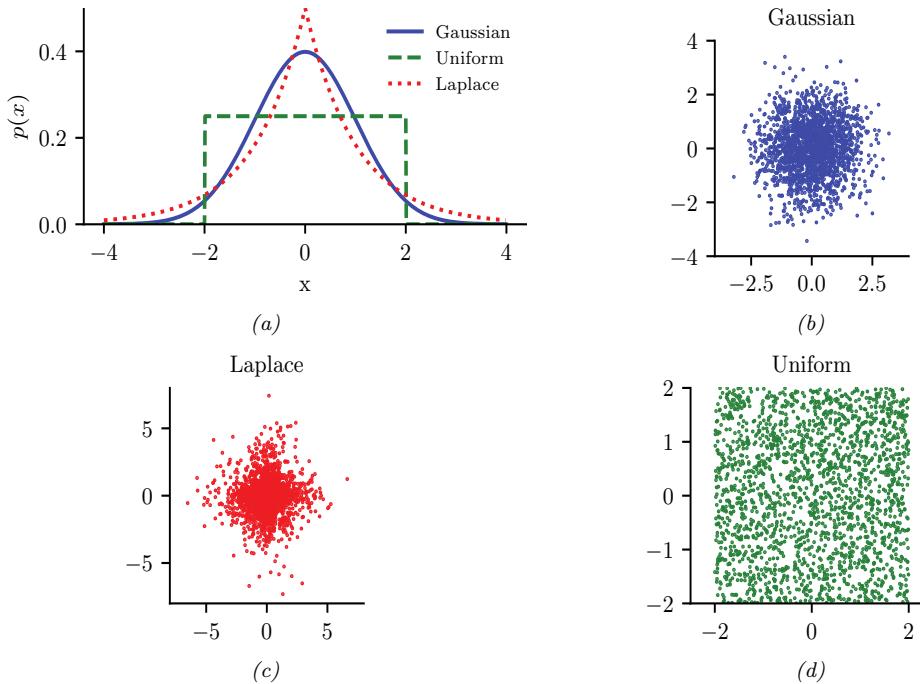


Figure 2.2: Illustration of Gaussian (blue), sub-Gaussian (uniform, green), and super-Gaussian (Laplace, red) distributions in 1d and 2d. Generated by [sub_super_gauss_plot.ipynb](#).

2.2.2.6 Sub-Gaussian and super-Gaussian distributions

There are two main variants of the Gaussian distribution, known as **super-Gaussian** or **leptokurtic** (“Lepto” is Greek for “narrow”) and **sub-Gaussian** or **platykurtic** (“Platy” is Greek for “broad”). These distributions differ in terms of their **kurtosis**, which is a measure of how heavy or light their tails are (i.e., how fast the density dies off to zero away from its mean). More precisely, the kurtosis is defined as

$$\text{kurt}(z) \triangleq \frac{\mu_4}{\sigma^4} = \frac{\mathbb{E}[(Z - \mu)^4]}{(\mathbb{E}[(Z - \mu)^2])^2} \quad (2.37)$$

where σ is the standard deviation, and μ_4 is the 4th **central moment**. (Thus $\mu_1 = \mu$ is the mean, and $\mu_2 = \sigma^2$ is the variance.) For a standard Gaussian, the kurtosis is 3, so some authors define the **excess kurtosis** as the kurtosis minus 3.

A super-Gaussian distribution (e.g., the Laplace) has positive excess kurtosis, and hence heavier tails than the Gaussian. A sub-Gaussian distribution, such as the uniform, has negative excess kurtosis, and hence lighter tails than the Gaussian. See Figure 2.2 for an illustration.

2.2.3 Continuous distributions on \mathbb{R}^+

In this section, we discuss some univariate distributions defined on the positive reals, $p(x)$ for $x \in \mathbb{R}^+$.

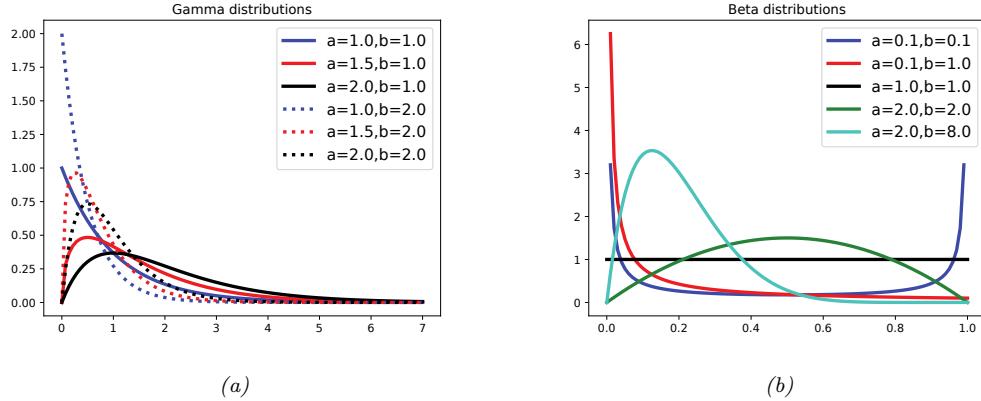


Figure 2.3: (a) Some gamma distributions. If $a \leq 1$, the mode is at 0; otherwise the mode is away from 0. As we increase the rate b , we reduce the horizontal scale, thus squeezing everything leftwards and upwards. Generated by `gamma_dist_plot.ipynb`. (b) Some beta distributions. If $a < 1$, we get a “spike” on the left, and if $b < 1$, we get a “spike” on the right. If $a = b = 1$, the distribution is uniform. If $a > 1$ and $b > 1$, the distribution is unimodal. Generated by `beta_dist_plot.ipynb`.

2.2.3.1 Gamma distribution

The **gamma distribution** is a flexible distribution for positive real valued rv's, $x > 0$. It is defined in terms of two parameters, called the shape $a > 0$ and the rate $b > 0$:

$$\text{Ga}(x|\text{shape} = a, \text{rate} = b) \triangleq \frac{b^a}{\Gamma(a)} x^{a-1} e^{-xb} \quad (2.38)$$

Sometimes the distribution is parameterized in terms of the rate a and the **scale** $s = 1/b$:

$$\text{Ga}(x|\text{shape} = a, \text{scale} = s) \triangleq \frac{1}{s^a \Gamma(a)} x^{a-1} e^{-x/s} \quad (2.39)$$

See Figure 2.3a for an illustration.

2.2.3.2 Exponential distribution

The **exponential distribution** is a special case of the gamma distribution and is defined by

$$\text{Expon}(x|\lambda) \triangleq \text{Ga}(x|\text{shape} = 1, \text{rate} = \lambda) \quad (2.40)$$

This distribution describes the times between events in a Poisson process, i.e., a process in which events occur continuously and independently at a constant average rate λ .

2.2.3.3 Chi-squared distribution

The **chi-squared distribution** is a special case of the gamma distribution and is defined by

$$\chi_\nu^2(x) \triangleq \text{Ga}(x|\text{shape} = \frac{\nu}{2}, \text{rate} = \frac{1}{2}) \quad (2.41)$$

where ν is called the degrees of freedom. This is the distribution of the sum of squared Gaussian random variables. More precisely, if $Z_i \sim \mathcal{N}(0, 1)$, and $S = \sum_{i=1}^{\nu} Z_i^2$, then $S \sim \chi_{\nu}^2$. Hence if $X \sim \mathcal{N}(0, \sigma^2)$ then $X^2 \sim \sigma^2 \chi_1^2$. Since $\mathbb{E}[\chi_1^2] = 1$ and $\mathbb{V}[\chi_1^2] = 2$, we have

$$\mathbb{E}[X^2] = \sigma^2, \mathbb{V}[X^2] = 2\sigma^4 \quad (2.42)$$

2.2.3.4 Inverse gamma

The **inverse gamma distribution**, denoted $Y \sim \text{IG}(a, b)$, is the distribution of $Y = 1/X$ assuming $X \sim \text{Ga}(a, b)$. This pdf is defined by

$$\text{IG}(x|\text{shape} = a, \text{scale} = b) \triangleq \frac{b^a}{\Gamma(a)} x^{-(a+1)} e^{-b/x} \quad (2.43)$$

The mean only exists if $a > 1$. The variance only exists if $a > 2$.

The **scaled inverse chi-squared distribution** is a reparameterization of the inverse gamma distribution:

$$\chi^{-2}(x|\nu, \sigma^2) = \text{IG}(x|\text{shape} = \frac{\nu}{2}, \text{scale} = \frac{\nu\sigma^2}{2}) \quad (2.44)$$

$$= \frac{1}{\Gamma(\nu/2)} \left(\frac{\nu\sigma^2}{2} \right)^{\nu/2} x^{-\frac{\nu}{2}-1} \exp\left(-\frac{\nu\sigma^2}{2x}\right) \quad (2.45)$$

The regular inverse chi-squared distribution, written $\chi_{\nu}^{-2}(x)$, is the special case where $\nu\sigma^2 = 1$ (i.e., $\sigma^2 = 1/\nu$). This corresponds to $\text{IG}(x|\text{shape} = \nu/2, \text{scale} = \frac{1}{2})$.

2.2.3.5 Pareto distribution

The **Pareto distribution** has the following pdf:

$$\text{Pareto}(x|m, \kappa) = \kappa m^{\kappa} \frac{1}{x^{(\kappa+1)}} \mathbb{I}(x \geq m) \quad (2.46)$$

See Figure 2.4(a) for some plots. We see that x must be greater than the minimum value m , but then the pdf rapidly decays after that. If we plot the distribution on a log-log scale, it forms the straight line $\log p(x) = -a \log x + \log(c)$, where $a = (\kappa + 1)$ and $c = \kappa m^{\kappa}$: see Figure 2.4(b) for an illustration.

When $m = 0$, the distribution has the form $p(x) = \kappa x^{-a}$. This is known as a **power law**. If $a = 1$, the distribution has the form $p(x) \propto 1/x$; if we interpret x as a frequency, this is called a $1/f$ function.

The Pareto distribution is useful for modeling the distribution of quantities that exhibit **heavy tails** or **long tails**, in which most values are small, but there are a few very large values. Many forms of data exhibit this property. ([ACL16] argue that this is because many datasets are generated by a variety of latent factors, which, when mixed together, naturally result in heavy tailed distributions.) We give some examples below.

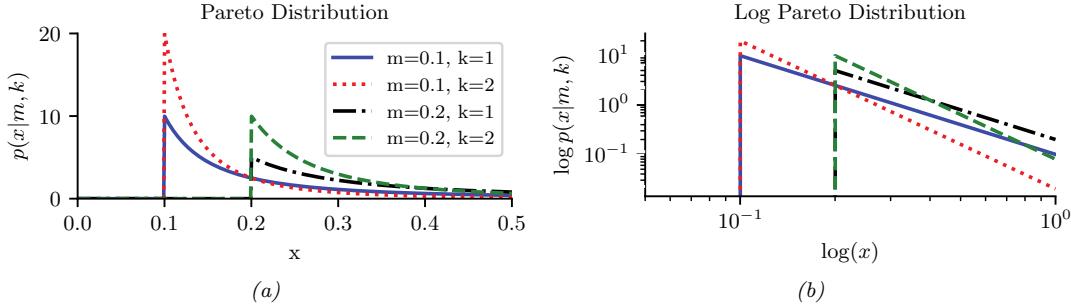


Figure 2.4: (a) The Pareto pdf $\text{Pareto}(x|k, m)$. (b) Same distribution on a log-log plot. Generated by `pareto_dist_plot.ipynb`.

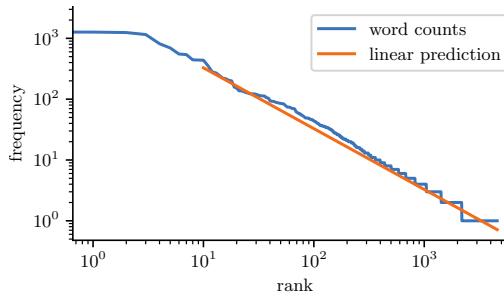


Figure 2.5: A log-log plot of the frequency vs the rank for the words in H. G. Wells' The Time Machine. Generated by `zipfs_law_plot.ipynb`. Adapted from a figure from [Zha+20a, Sec 8.3].

Modeling wealth distributions

The Pareto distribution is named after the Italian economist and sociologist Vilfredo Pareto. He created it in order to model the distribution of wealth across different countries. Indeed, in economics, the parameter κ is called the **Pareto index**. If we set $\kappa = 1.16$, we recover the **80-20 rule**, which states that 80% of the wealth of a society is held by 20% of the population.²

Zipf's law

Zipf's law says that the most frequent word in a language (such as “the”) occurs approximately twice as often as the second most frequent word (“of”), which occurs twice as often as the fourth most frequent word, etc. This corresponds to a Pareto distribution of the form

$$p(x = r) \propto \kappa r^{-\alpha} \tag{2.47}$$

2. In fact, wealth distributions are even more skewed than this. For example, as of 2014, 80 billionaires now have as much wealth as 3.5 billion people! (Source: <http://www.pbs.org/newshour/making-sense/wealthiest-getting-wealthier-lobbying-lot/>.) Such extreme income inequality exists in many plutocratic countries, including the USA (see e.g., [HP10]).

where r is the rank of word x when sorted by frequency, and κ and a are constants. If we set $a = 1$, we recover Zipf's law.³ Thus Zipf's law predicts that if we plot the log frequency of words vs their log rank, we will get a straight line with slope -1 . This is in fact true, as illustrated in Figure 2.5.⁴ See [Ada00] for further discussion of Zipf's law, and Section 2.6.2 for a discussion of language models.

2.2.4 Continuous distributions on $[0, 1]$

In this section, we discuss some univariate distributions defined on the $[0, 1]$ interval.

2.2.4.1 Beta distribution

The **beta distribution** has support over the interval $[0, 1]$ and is defined as follows:

$$\text{Beta}(x|a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} \quad (2.48)$$

where $B(a, b)$ is the **beta function**. We require $a, b > 0$ to ensure the distribution is integrable (i.e., to ensure $B(a, b)$ exists). If $a = b = 1$, we get the uniform distribution. If a and b are both less than 1, we get a bimodal distribution with "spikes" at 0 and 1; if a and b are both greater than 1, the distribution is unimodal. See Figure 2.3b.

2.2.5 Multivariate continuous distributions

In this section, we summarize some other widely used multivariate continuous distributions.

2.2.5.1 Multivariate normal (Gaussian)

The **multivariate normal (MVN)**, also called the **multivariate Gaussian**, is by far the most widely used multivariate distribution. As such, the whole of Section 2.3 is dedicated to it.

2.2.5.2 Multivariate Student distribution

One problem with Gaussians is that they are sensitive to outliers. Fortunately, we can easily extend the Student distribution, discussed in Main Section 2.2.3, to D dimensions. In particular, the pdf of the **multivariate Student distribution** is given by

$$\mathcal{T}_\nu(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{Z} \left[1 + \frac{1}{\nu} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right]^{-(\frac{\nu+D}{2})} \quad (2.49)$$

$$Z = \frac{\Gamma(\nu/2)}{\Gamma(\nu/2 + D/2)} \frac{\nu^{D/2} \pi^{D/2}}{|\boldsymbol{\Sigma}|^{-1/2}} \quad (2.50)$$

where $\boldsymbol{\Sigma}$ is called the scale matrix.

3. For example, $p(x=2) = \kappa 2^{-1} = 2\kappa 4^{-1} = 2p(x=4)$.

4. We remove the first 10 words from the plot, since they don't fit the prediction as well.

The Student has fatter tails than a Gaussian. The smaller ν is, the fatter the tails. As $\nu \rightarrow \infty$, the distribution tends towards a Gaussian. The distribution has these properties:

$$\text{mean} = \boldsymbol{\mu}, \text{ mode} = \boldsymbol{\mu}, \text{cov} = \frac{\nu}{\nu - 2} \boldsymbol{\Sigma} \quad (2.51)$$

The mean is only well defined (finite) if $\nu > 1$. Similarly, the covariance is only well defined if $\nu > 2$.

2.2.5.3 Circular normal (von Mises Fisher) distribution

Sometimes data lives on the unit sphere, rather than being any point in Euclidean space. For example, any D dimensional vector that is ℓ_2 -normalized lives on the unit $(D - 1)$ sphere embedded in \mathbb{R}^D .

There is an extension of the Gaussian distribution that is suitable for such angular data, known as the **von Mises-Fisher** distribution, or the **circular normal** distribution. It has the following pdf:

$$\text{vMF}(\mathbf{x}|\boldsymbol{\mu}, \kappa) \triangleq \frac{1}{Z} \exp(\kappa \boldsymbol{\mu}^\top \mathbf{x}) \quad (2.52)$$

$$Z = \frac{(2\pi)^{D/2} I_{D/2-1}(\kappa)}{\kappa^{D/2-1}} \quad (2.53)$$

where $\boldsymbol{\mu}$ is the mean (with $\|\boldsymbol{\mu}\| = 1$), $\kappa \geq 0$ is the concentration or precision parameter (analogous to $1/\sigma$ for a standard Gaussian), and Z is the normalization constant, with $I_r(\cdot)$ being the modified Bessel function of the first kind and order r . The vMF is like a spherical multivariate Gaussian, parameterized by **cosine distance** instead of Euclidean distance.

The vMF distribution can be used inside of a mixture model to cluster ℓ_2 -normalized vectors, as an alternative to using a Gaussian mixture model [Ban+05]. If $\kappa \rightarrow 0$, this reduces to the spherical K-means algorithm. It can also be used inside of an admixture model (Main Section 28.4.2); this is called the spherical topic model [Rei+10].

If $D = 2$, an alternative is to use the **von Mises** distribution on the unit circle, which has the form

$$\text{vMF}(x|\mu, \kappa) = \frac{1}{Z} \exp(\kappa \cos(x - \mu)) \quad (2.54)$$

$$Z = 2\pi I_0(\kappa) \quad (2.55)$$

2.2.5.4 Matrix normal distribution (MN)

The **matrix normal** distribution is defined by the following probability density function over matrices $\mathbf{X} \in \mathbb{R}^{n \times p}$:

$$\mathcal{MN}(\mathbf{X}|\mathbf{M}, \mathbf{U}, \mathbf{V}) \triangleq \frac{|\mathbf{V}|^{n/2}}{2\pi^{np/2} |\mathbf{U}|^{p/2}} \exp \left\{ -\frac{1}{2} \text{tr} [(\mathbf{X} - \mathbf{M})^\top \mathbf{U}^{-1} (\mathbf{X} - \mathbf{M}) \mathbf{V}] \right\} \quad (2.56)$$

where $\mathbf{M} \in \mathbb{R}^{n \times p}$ is the mean value of \mathbf{X} , $\mathbf{U} \in \mathcal{S}_{++}^{n \times n}$ is the covariance among rows, and $\mathbf{V} \in \mathcal{S}_{++}^{p \times p}$ is the precision among columns. It can be seen that

$$\text{vec}(\mathbf{X}) \sim \mathcal{N}(\text{vec}(\mathbf{M}), \mathbf{V}^{-1} \otimes \mathbf{U}). \quad (2.57)$$

Note that there is another version of the definition of the matrix normal distribution using the column-covariance matrix $\tilde{\mathbf{V}} = \mathbf{V}^{-1}$ instead of \mathbf{V} , which leads to the density

$$\frac{1}{2\pi^{np/2}|\mathbf{U}|^{p/2}|\tilde{\mathbf{V}}|^{n/2}} \exp \left\{ -\frac{1}{2} \text{tr} [(\mathbf{X} - \mathbf{M})^\top \mathbf{U}^{-1} (\mathbf{X} - \mathbf{M}) \tilde{\mathbf{V}}^{-1}] \right\}. \quad (2.58)$$

These two versions of definition are obviously equivalent, but we will see that the definition we adopt in Equation (2.56) will lead to a neat update of the posterior distribution (just as the precision matrix is more convenient to use than the covariance matrix in analyzing the posterior of the multivariate normal distribution with a conjugate prior).

2.2.5.5 Wishart distribution

The **Wishart** distribution is the generalization of the gamma distribution to positive definite matrices. Press [Pre05, p107] has said, “The Wishart distribution ranks next to the normal distribution in order of importance and usefulness in multivariate statistics”. We will mostly use it to model our uncertainty when estimating covariance matrices (see Section 3.4.4).

The pdf of the Wishart is defined as follows:

$$\text{Wi}(\boldsymbol{\Sigma}|\mathbf{S}, \nu) \triangleq \frac{1}{Z} |\boldsymbol{\Sigma}|^{(\nu-D-1)/2} \exp \left(-\frac{1}{2} \text{tr}(\mathbf{S}^{-1}\boldsymbol{\Sigma}) \right) \quad (2.59)$$

$$Z \triangleq |\mathbf{S}|^{-\nu/2} 2^{\nu D/2} \Gamma_D(\nu/2) \quad (2.60)$$

Here ν is called the “degrees of freedom” and \mathbf{S} is the “scale matrix”. (We shall get more intuition for these parameters shortly.) The normalization constant only exists (and hence the pdf is only well defined) if $\nu > D - 1$.

The distribution has these properties:

$$\text{mean} = \nu \mathbf{S}, \text{ mode} = (\nu - D - 1) \mathbf{S} \quad (2.61)$$

Note that the mode only exists if $\nu > D + 1$.

If $D = 1$, the Wishart reduces to the gamma distribution:

$$\text{Wi}(\lambda|s^{-1}, \nu) = \text{Ga}(\lambda|\text{shape} = \frac{\nu}{2}, \text{rate} = \frac{1}{2s}) \quad (2.62)$$

If $s = 2$, this reduces to the chi-squared distribution.

There is an interesting connection between the Wishart distribution and the Gaussian. In particular, let $\mathbf{x}_n \sim \mathcal{N}(0, \boldsymbol{\Sigma})$. One can show that the scatter matrix, $\mathbf{S} = \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top$, has a Wishart distribution: $\mathbf{S} \sim \text{Wi}(\boldsymbol{\Sigma}, N)$.

2.2.5.6 Inverse Wishart distribution

If $\lambda \sim \text{Ga}(a, b)$, then that $\frac{1}{\lambda} \sim \text{IG}(a, b)$. Similarly, if $\boldsymbol{\Sigma}^{-1} \sim \text{Wi}(\mathbf{S}^{-1}, \nu)$ then $\boldsymbol{\Sigma} \sim \text{IW}(\mathbf{S}, \nu)$, where IW is the **inverse Wishart**, the multidimensional generalization of the inverse gamma. It is defined as follows, for $\nu > D - 1$ and $\mathbf{S} \succ 0$:

$$\text{IW}(\boldsymbol{\Sigma}|\mathbf{S}^{-1}, \nu) = \frac{1}{Z} |\boldsymbol{\Sigma}|^{-(\nu+D+1)/2} \exp \left(-\frac{1}{2} \text{tr}(\mathbf{S}\boldsymbol{\Sigma}^{-1}) \right) \quad (2.63)$$

$$Z_{\text{IW}} = |\mathbf{S}|^{\nu/2} 2^{\nu D/2} \Gamma_D(\nu/2) \quad (2.64)$$

One can show that the distribution has these properties:

$$\text{mean} = \frac{\mathbf{S}}{\nu - D - 1}, \quad \text{mode} = \frac{\mathbf{S}}{\nu + D + 1} \quad (2.65)$$

If $D = 1$, this reduces to the inverse gamma:

$$\text{IW}(\sigma^2 | s^{-1}, \nu) = \text{IG}(\sigma^2 | \nu/2, s/2) \quad (2.66)$$

If $s = 1$, this reduces to the inverse chi-squared distribution.

2.2.5.7 Dirichlet distribution

A multivariate generalization of the beta distribution is the **Dirichlet**⁵ distribution, which has support over the **probability simplex**, defined by

$$S_K = \{\mathbf{x} : 0 \leq x_k \leq 1, \sum_{k=1}^K x_k = 1\} \quad (2.67)$$

The pdf is defined as follows:

$$\text{Dir}(\mathbf{x} | \boldsymbol{\alpha}) \triangleq \frac{1}{B(\boldsymbol{\alpha})} \prod_{k=1}^K x_k^{\alpha_k - 1} \mathbb{I}(\mathbf{x} \in S_K) \quad (2.68)$$

where $B(\boldsymbol{\alpha})$ is the multivariate beta function,

$$B(\boldsymbol{\alpha}) \triangleq \frac{\prod_{k=1}^K \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^K \alpha_k)} \quad (2.69)$$

Figure 2.6 shows some plots of the Dirichlet when $K = 3$. We see that $\alpha_0 = \sum_k \alpha_k$ controls the strength of the distribution (how peaked it is), and the α_k control where the peak occurs. For example, $\text{Dir}(1, 1, 1)$ is a uniform distribution, $\text{Dir}(2, 2, 2)$ is a broad distribution centered at $(1/3, 1/3, 1/3)$, and $\text{Dir}(20, 20, 20)$ is a narrow distribution centered at $(1/3, 1/3, 1/3)$. $\text{Dir}(3, 3, 20)$ is an asymmetric distribution that puts more density in one of the corners. If $\alpha_k < 1$ for all k , we get “spikes” at the corners of the simplex. Samples from the distribution when $\alpha_k < 1$ are sparse, as shown in Figure 2.7.

For future reference, here are some useful properties of the Dirichlet distribution:

$$\mathbb{E}[x_k] = \frac{\alpha_k}{\alpha_0}, \quad \text{mode}[x_k] = \frac{\alpha_k - 1}{\alpha_0 - K}, \quad \mathbb{V}[x_k] = \frac{\alpha_k(\alpha_0 - \alpha_k)}{\alpha_0^2(\alpha_0 + 1)} \quad (2.70)$$

where $\alpha_0 = \sum_k \alpha_k$.

Often we use a symmetric Dirichlet prior of the form $\alpha_k = \alpha/K$. In this case, we have $\mathbb{E}[x_k] = 1/K$, and $\mathbb{V}[x_k] = \frac{K-1}{K^2(\alpha+1)}$. So we see that increasing α increases the precision (decreases the variance) of the distribution.

5. Johann Dirichlet was a German mathematician, 1805–1859.

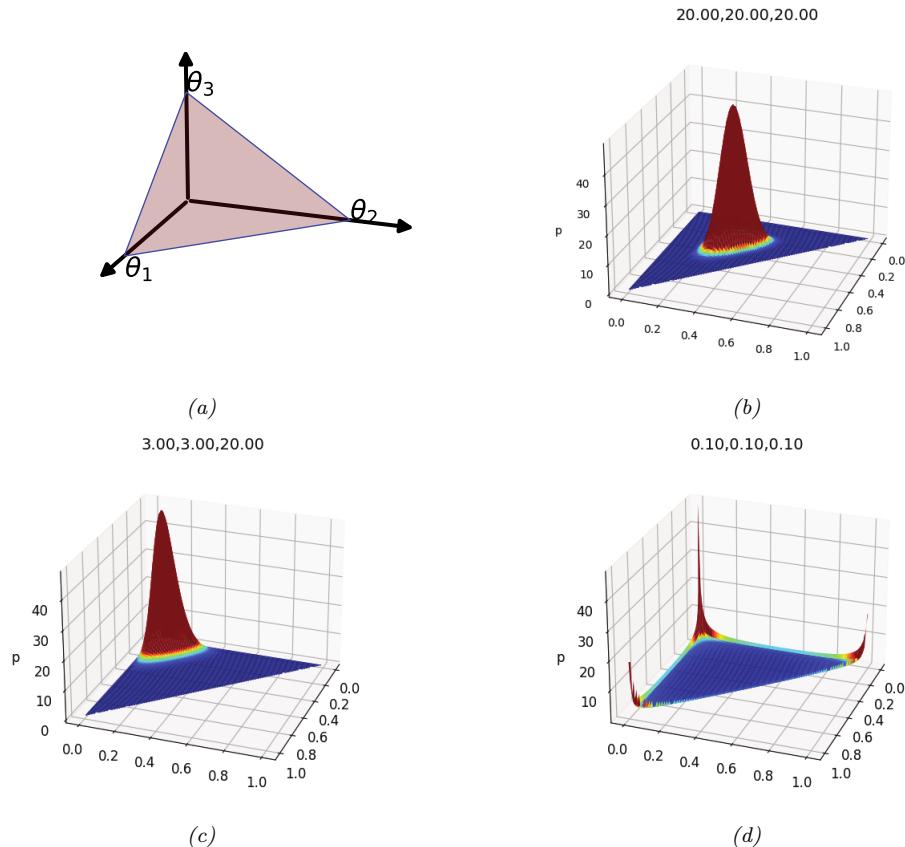


Figure 2.6: (a) The Dirichlet distribution when $K = 3$ defines a distribution over the simplex, which can be represented by the triangular surface. Points on this surface satisfy $0 \leq \theta_c \leq 1$ and $\sum_{c=1}^3 \theta_c = 1$. Generated by `dirichlet_3d_triangle_plot.ipynb`. (b) Plot of the Dirichlet density for $\alpha = (20, 20, 20)$. (c) Plot of the Dirichlet density for $\alpha = (3, 3, 20)$. (d) Plot of the Dirichlet density for $\alpha = (0.1, 0.1, 0.1)$. Generated by `dirichlet_3d_spiky_plot.ipynb`.

The Dirichlet distribution is useful for distinguishing aleatoric (data) uncertainty from epistemic uncertainty. To see this, consider a 3-sided die. If we know that each outcome is equally likely, we can use a “peaky” symmetric Dirichlet, such as $\text{Dir}(20, 20, 20)$, shown in Figure 2.6(b); this reflects the fact that we are sure the outcomes will be unpredictable. By contrast, if we are not sure what the outcomes will be like (e.g., it could be a biased die), then we can use a “flat” symmetric Dirichlet, such as $\text{Dir}(1, 1, 1)$, which can generate a wide range of possible outcome distributions. We can make the Dirichlet distribution be conditional on inputs, resulting in what is called a **prior network** [MG18], since it encodes $p(\pi|\mathbf{x})$ (output is a distribution) rather than $p(y|\mathbf{x})$ (output is a label).

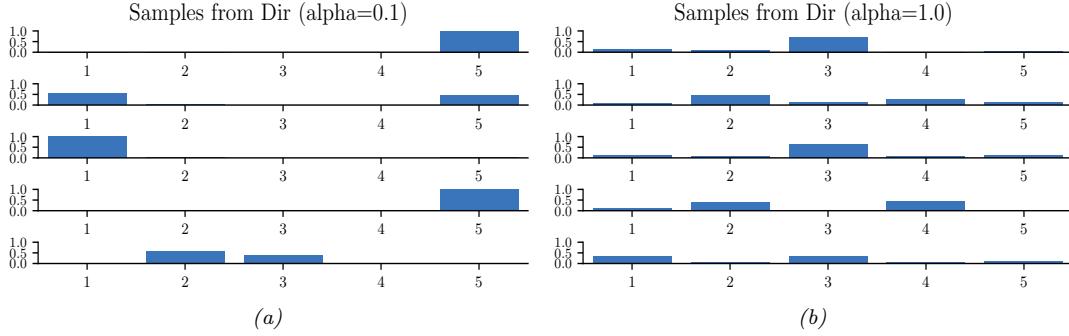


Figure 2.7: Samples from a 5-dimensional symmetric Dirichlet distribution for different parameter values. (a) $\boldsymbol{\alpha} = (0.1, \dots, 0.1)$. This results in very sparse distributions, with many 0s. (b) $\boldsymbol{\alpha} = (1, \dots, 1)$. This results in more uniform (and dense) distributions. Generated by `dirichlet_samples_plot.ipynb`.

2.3 Gaussian joint distributions

The most widely used joint probability distribution for continuous random variables is the **multivariate Gaussian** or **multivariate normal (MVN)**. The popularity is partly because this distribution is mathematically convenient, but also because the Gaussian assumption is fairly reasonable in many cases. Indeed, the Gaussian is the distribution with maximum entropy subject to having specified first and second moments (Section 2.4.7). In view of its importance, this section discusses the Gaussian distribution in detail.

2.3.1 The multivariate normal

In this section, we discuss the multivariate Gaussian or multivariate normal in detail.

2.3.1.1 Definition

The MVN density is defined by the following:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \triangleq \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp \left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right] \quad (2.71)$$

where $\boldsymbol{\mu} = \mathbb{E}[\mathbf{x}] \in \mathbb{R}^D$ is the mean vector, and $\boldsymbol{\Sigma} = \text{Cov}[\mathbf{x}]$ is the $D \times D$ covariance matrix. The normalization constant $Z = (2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}$ just ensures that the pdf integrates to 1. The expression inside the exponential (ignoring the factor of -0.5) is the squared **Mahalanobis distance** between the data vector \mathbf{x} and the mean vector $\boldsymbol{\mu}$, given by

$$d_{\boldsymbol{\Sigma}}(\mathbf{x}, \boldsymbol{\mu})^2 = (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \quad (2.72)$$

In 2d, the MVN is known as the **bivariate Gaussian** distribution. Its pdf can be represented as $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\mathbf{x} \in \mathbb{R}^2$, $\boldsymbol{\mu} \in \mathbb{R}^2$ and

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & \sigma_{12}^2 \\ \sigma_{21}^2 & \sigma_2^2 \end{pmatrix} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \quad (2.73)$$

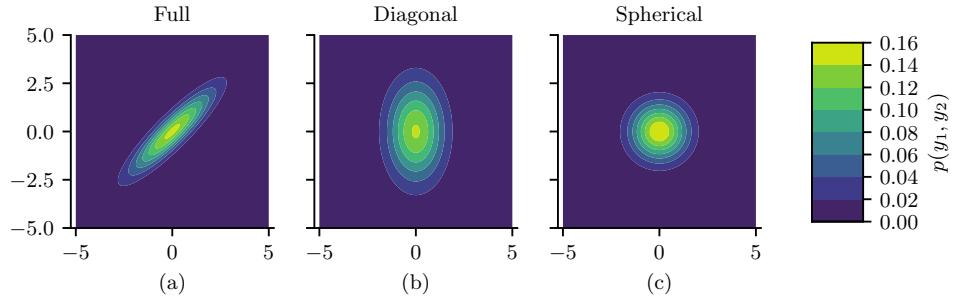


Figure 2.8: Visualization of a 2d Gaussian density in terms of level sets of constant probability density. (a) A full covariance matrix has elliptical contours. (b) A diagonal covariance matrix is an axis aligned ellipse. (c) A spherical covariance matrix has a circular shape. Generated by [gauss_plot_2d.ipynb](#).

where the correlation coefficient is given by $\rho \triangleq \frac{\sigma_{12}^2}{\sigma_1 \sigma_2}$.

Figure 2.8 plots some MVN densities in 2d for three different kinds of covariance matrices. A **full covariance matrix** has $D(D + 1)/2$ parameters, where we divide by 2 since Σ is symmetric. A **diagonal covariance matrix** has D parameters, and has 0s in the off-diagonal terms. A **spherical covariance matrix**, also called **isotropic covariance matrix**, has the form $\Sigma = \sigma^2 \mathbf{I}_D$, so it only has one free parameter, namely σ^2 .

2.3.1.2 Gaussian shells

Multivariate Gaussians can behave rather counterintuitively in high dimensions. In particular, we can ask: if we draw samples $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_D)$, where D is the number of dimensions, where do we expect most of the \mathbf{x} to lie? Since the peak (mode) of the pdf is at the origin, it is natural to expect most samples to be near the origin. However, in high dimensions, the typical set of a Gaussian is a thin shell or annulus with a distance from origin given by $r = \sigma\sqrt{D}$ and a thickness of $O(\sigma D^{1/4})$. The intuitive reason for this is as follows: although the density decays as $e^{-r^2/2}$, meaning density decreases from the origin, the volume of a sphere grows as r^D , meaning volume increases from the origin, and since mass is density times volume, the majority of points end up in this annulus where these two terms “balance out”. This is called the “**Gaussian soap bubble**” phenomenon, and is illustrated in Figure 2.9.⁶

To see why the typical set for a Gaussian is concentrated in a thin annulus at radius \sqrt{D} , consider the squared distance of a point \mathbf{x} from the origin, $d(\mathbf{x}) = \sqrt{\sum_{i=1}^D x_i^2}$, where $x_i \sim \mathcal{N}(0, 1)$. The expected squared distance is given by $\mathbb{E}[d^2] = \sum_{i=1}^D \mathbb{E}[x_i^2] = D$, and the variance of the squared distance is given by $\mathbb{V}[d^2] = \sum_{i=1}^D \mathbb{V}[x_i^2] = D$. As D grows, the coefficient of variation (i.e., the SD relative to the mean) goes to zero:

$$\lim_{D \rightarrow \infty} \frac{\text{std}[d^2]}{\mathbb{E}[d^2]} = \lim_{D \rightarrow \infty} \frac{\sqrt{D}}{D} = 0 \quad (2.74)$$

⁶. For a more detailed explanation, see this blog post by Ferenc Huszar: <https://www.inference.vc/high-dimensional-gaussian-distributions-are-soap-bubble/>.

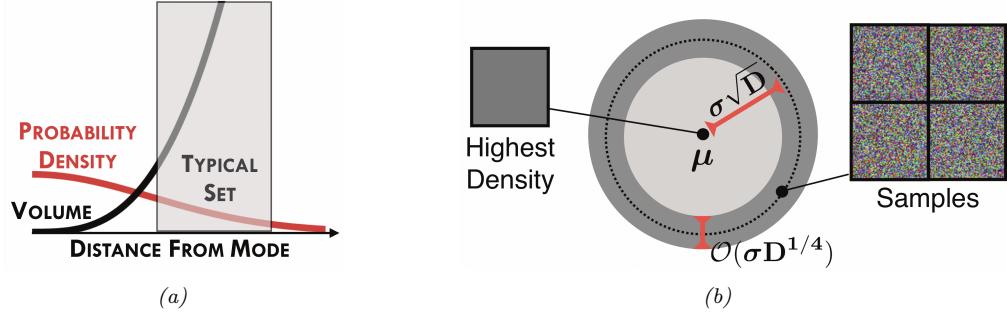


Figure 2.9: (a) Cartoon illustration of why the typical set of a Gaussian is not centered at the mode of the distribution. (b) Illustration of the typical set of a Gaussian, which is concentrated in a thin annulus of thickness $\sigma D^{1/4}$ and distance $\sigma D^{1/2}$ from the origin. We also show an image with the highest density (the all gray image on the left), as well as some high probability samples (the speckle noise images on the right). From Figure 1 of [Nal+19a]. Used with kind permission of Eric Nalisnick.

Thus the expected square distance concentrates around D , so the expected distance concentrates around $\mathbb{E}[d(\mathbf{x})] = \sqrt{D}$. See [Ver18] for a more rigorous proof, and Section 5.2.3 for a discussion of typical sets.

To see what this means in the context of images, in Figure 2.9b, we show some grayscale images that are sampled from a Gaussian of the form $\mathcal{N}(\boldsymbol{\mu}, \sigma^2 \mathbf{I})$, where $\boldsymbol{\mu}$ corresponds to the all-gray image. However, it is extremely unlikely that randomly sampled images would be close to all-gray, as shown in the figure.

2.3.1.3 Marginals and conditionals of an MVN

Let us partition our vector of random variables \mathbf{x} into two parts, \mathbf{x}_1 and \mathbf{x}_2 , so

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix} \quad (2.75)$$

The marginals of this distribution are given by the following (see Section 2.3.1.5 for the proof):

$$p(\mathbf{x}_1) = \int \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{x}_2 \triangleq \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_1^m, \boldsymbol{\Sigma}_1^m) = \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \quad (2.76)$$

$$p(\mathbf{x}_2) = \int \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{x}_1 \triangleq \mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_2^m, \boldsymbol{\Sigma}_2^m) = \mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \quad (2.77)$$

The conditional distributions can be shown to have the following form (see Section 2.3.1.5 for the proof):

$$p(\mathbf{x}_1|\mathbf{x}_2) = \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_{1|2}^c, \boldsymbol{\Sigma}_{1|2}^c) = \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2), \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}) \quad (2.78)$$

$$p(\mathbf{x}_2|\mathbf{x}_1) = \mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_{2|1}^c, \boldsymbol{\Sigma}_{2|1}^c) = \mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \boldsymbol{\mu}_1), \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}) \quad (2.79)$$

Note that the posterior mean of $p(\mathbf{x}_1|\mathbf{x}_2)$ is a linear function of \mathbf{x}_2 , but the posterior covariance is independent of \mathbf{x}_2 ; this is a peculiar property of Gaussian distributions.

2.3.1.4 Information (canonical) form

It is common to parameterize the MVN in terms of the mean vector μ and the covariance matrix Σ . However, for reasons which are explained in Section 2.4.2.5, it is sometimes useful to represent the Gaussian distribution using **canonical parameters** or **natural parameters**, defined as

$$\Lambda \triangleq \Sigma^{-1}, \quad \eta \triangleq \Sigma^{-1}\mu \quad (2.80)$$

The matrix $\Lambda = \Sigma^{-1}$ is known as the **precision matrix**, and the vector η is known as the **precision-weighted mean**. We can convert back to the more familiar **moment parameters** using

$$\mu = \Lambda^{-1}\eta, \quad \Sigma = \Lambda^{-1} \quad (2.81)$$

Hence we can write the MVN in **canonical form** (also called **information form**) as follows:

$$\mathcal{N}_c(\mathbf{x}|\eta, \Lambda) \triangleq c \exp\left(\mathbf{x}^\top \eta - \frac{1}{2} \mathbf{x}^\top \Lambda \mathbf{x}\right) \quad (2.82)$$

$$c \triangleq \frac{\exp(-\frac{1}{2} \eta^\top \Lambda^{-1} \eta)}{(2\pi)^{D/2} \sqrt{\det(\Lambda^{-1})}} \quad (2.83)$$

where we use the notation $\mathcal{N}_c()$ to distinguish it from the standard parameterization $\mathcal{N}()$. For more information on moment and natural parameters, see Section 2.4.2.5.

It is also possible to derive the marginalization and conditioning formulas in information form (see Section 2.3.1.6 for the derivation). For the marginals we have

$$p(\mathbf{x}_1) = \mathcal{N}_c(\mathbf{x}_1|\eta_1^m, \Lambda_1^m) = \mathcal{N}_c(\mathbf{x}_1|\eta_1 - \Lambda_{12}\Lambda_{22}^{-1}\eta_2, \Lambda_{11} - \Lambda_{12}\Lambda_{22}^{-1}\Lambda_{21}) \quad (2.84)$$

$$p(\mathbf{x}_2) = \mathcal{N}_c(\mathbf{x}_2|\eta_2^m, \Lambda_2^m) = \mathcal{N}_c(\mathbf{x}_2|\eta_2 - \Lambda_{21}\Lambda_{11}^{-1}\eta_1, \Lambda_{22} - \Lambda_{21}\Lambda_{11}^{-1}\Lambda_{12}) \quad (2.85)$$

For the conditionals we have

$$p(\mathbf{x}_1|\mathbf{x}_2) = \mathcal{N}_c(\mathbf{x}_1|\eta_{1|2}^c, \Lambda_{1|2}^c) = \mathcal{N}_c(\mathbf{x}_1|\eta_1 - \Lambda_{12}\mathbf{x}_2, \Lambda_{11}) \quad (2.86)$$

$$p(\mathbf{x}_2|\mathbf{x}_1) = \mathcal{N}_c(\mathbf{x}_2|\eta_{2|1}^c, \Lambda_{2|1}^c) = \mathcal{N}_c(\mathbf{x}_2|\eta_2 - \Lambda_{21}\mathbf{x}_1, \Lambda_{22}) \quad (2.87)$$

Thus we see that marginalization is easier in moment form, and conditioning is easier in information form.

2.3.1.5 Derivation: moment form

In this section, we derive Equation (2.77) and Equation (2.78) for marginalizing and conditioning an MVN in moment form.

Before we dive in, we need to introduce the following result, for the **inverse of a partitioned matrix** of the form

$$\mathbf{M} = \begin{pmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{pmatrix} \quad (2.88)$$

where we assume \mathbf{E} and \mathbf{H} are invertible. One can show (see e.g., [Mur22, Sec 7.3.2] for the proof) that

$$\mathbf{M}^{-1} = \begin{pmatrix} (\mathbf{M}/\mathbf{H})^{-1} & -(\mathbf{M}/\mathbf{H})^{-1}\mathbf{F}\mathbf{H}^{-1} \\ -\mathbf{H}^{-1}\mathbf{G}(\mathbf{M}/\mathbf{H})^{-1} & \mathbf{H}^{-1} + \mathbf{H}^{-1}\mathbf{G}(\mathbf{M}/\mathbf{H})^{-1}\mathbf{F}\mathbf{H}^{-1} \end{pmatrix} \quad (2.89)$$

$$= \begin{pmatrix} \mathbf{E}^{-1} + \mathbf{E}^{-1}\mathbf{F}(\mathbf{M}/\mathbf{E})^{-1}\mathbf{G}\mathbf{E}^{-1} & -\mathbf{E}^{-1}\mathbf{F}(\mathbf{M}/\mathbf{E})^{-1} \\ -(\mathbf{M}/\mathbf{E})^{-1}\mathbf{G}\mathbf{E}^{-1} & (\mathbf{M}/\mathbf{E})^{-1} \end{pmatrix} \quad (2.90)$$

where

$$\mathbf{M}/\mathbf{H} \triangleq \mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G} \quad (2.91)$$

$$\mathbf{M}/\mathbf{E} \triangleq \mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F} \quad (2.92)$$

We say that \mathbf{M}/\mathbf{H} is the **Schur complement** of \mathbf{M} wrt \mathbf{H} , and \mathbf{M}/\mathbf{E} is the Schur complement of \mathbf{M} wrt \mathbf{E} .

From the above, we also have the following important result, known as the **matrix inversion lemma** or the **Sherman-Morrison-Woodbury formula**:

$$(\mathbf{M}/\mathbf{H})^{-1} = (\mathbf{E} - \mathbf{F}\mathbf{H}^{-1}\mathbf{G})^{-1} = \mathbf{E}^{-1} + \mathbf{E}^{-1}\mathbf{F}(\mathbf{H} - \mathbf{G}\mathbf{E}^{-1}\mathbf{F})^{-1}\mathbf{G}\mathbf{E}^{-1} \quad (2.93)$$

Now we return to the derivation of the MVN conditioning equation. Let us factor the joint $p(\mathbf{x}_1, \mathbf{x}_2)$ as $p(\mathbf{x}_2)p(\mathbf{x}_1|\mathbf{x}_2)$ as follows:

$$p(\mathbf{x}_1, \mathbf{x}_2) \propto \exp \left\{ -\frac{1}{2} \begin{pmatrix} \mathbf{x}_1 - \boldsymbol{\mu}_1 \\ \mathbf{x}_2 - \boldsymbol{\mu}_2 \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{x}_1 - \boldsymbol{\mu}_1 \\ \mathbf{x}_2 - \boldsymbol{\mu}_2 \end{pmatrix} \right\} \quad (2.94)$$

Using the equation for the inverse of a block structured matrix, the above exponent becomes

$$p(\mathbf{x}_1, \mathbf{x}_2) \propto \exp \left\{ -\frac{1}{2} \begin{pmatrix} \mathbf{x}_1 - \boldsymbol{\mu}_1 \\ \mathbf{x}_2 - \boldsymbol{\mu}_2 \end{pmatrix}^\top \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ -\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} & \mathbf{I} \end{pmatrix} \begin{pmatrix} (\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_{22}^{-1} \end{pmatrix} \right\} \quad (2.95)$$

$$\times \begin{pmatrix} \mathbf{I} & -\boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 - \boldsymbol{\mu}_1 \\ \mathbf{x}_2 - \boldsymbol{\mu}_2 \end{pmatrix} \quad (2.96)$$

$$= \exp \left\{ -\frac{1}{2} (\mathbf{x}_1 - \boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2))^\top (\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22})^{-1} \right. \quad (2.97)$$

$$\left. (\mathbf{x}_1 - \boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2)) \right\} \times \exp \left\{ -\frac{1}{2} (\mathbf{x}_2 - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_{22}^{-1} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \right\} \quad (2.98)$$

This is of the form

$$\exp(\text{quadratic form in } \mathbf{x}_1, \mathbf{x}_2) \times \exp(\text{quadratic form in } \mathbf{x}_2) \quad (2.99)$$

Hence we have successfully factorized the joint as

$$p(\mathbf{x}_1, \mathbf{x}_2) = p(\mathbf{x}_1|\mathbf{x}_2)p(\mathbf{x}_2) \quad (2.100)$$

$$= \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2})\mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \quad (2.101)$$

where

$$\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2) \quad (2.102)$$

$$\boldsymbol{\Sigma}_{1|2} = \boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22} \triangleq \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} \quad (2.103)$$

where $\boldsymbol{\Sigma}/\boldsymbol{\Sigma}_{22}$ is as the Schur complement of $\boldsymbol{\Sigma}$ wrt $\boldsymbol{\Sigma}_{22}$.

2.3.1.6 Derivation: information form

In this section, we derive Equation (2.85) and Equation (2.86) for marginalizing and conditioning an MVN in information form.

First we derive the conditional formula.⁷ Let us partition the information form parameters as follows:

$$\boldsymbol{\eta} = \begin{pmatrix} \boldsymbol{\eta}_1 \\ \boldsymbol{\eta}_2 \end{pmatrix}, \quad \boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix} \quad (2.104)$$

We can now write the joint log probability of $\boldsymbol{x}_1, \boldsymbol{x}_2$ as

$$\ln p(\boldsymbol{x}_1, \boldsymbol{x}_2) = -\frac{1}{2} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix} + \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix}^\top \begin{pmatrix} \boldsymbol{\eta}_1 \\ \boldsymbol{\eta}_2 \end{pmatrix} + \text{const.} \quad (2.105)$$

$$\begin{aligned} &= -\frac{1}{2} \boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{11} \boldsymbol{x}_1 - \frac{1}{2} \boldsymbol{x}_2^\top \boldsymbol{\Lambda}_{22} \boldsymbol{x}_2 - \frac{1}{2} \boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{12} \boldsymbol{x}_2 - \frac{1}{2} \boldsymbol{x}_2^\top \boldsymbol{\Lambda}_{21} \boldsymbol{x}_1 \\ &\quad + \boldsymbol{x}_1^\top \boldsymbol{\eta}_1 + \boldsymbol{x}_2^\top \boldsymbol{\eta}_2 + \text{const.} \end{aligned} \quad (2.106)$$

where the constant term does not depend on \boldsymbol{x}_1 or \boldsymbol{x}_2 .

To calculate the parameters of the conditional distribution $p(\boldsymbol{x}_1|\boldsymbol{x}_2)$, we fix the value of \boldsymbol{x}_2 and collect the terms which are quadratic in \boldsymbol{x}_1 for the conditional precision and then linear in \boldsymbol{x}_1 for the conditional precision-weighted mean. The terms which are quadratic in \boldsymbol{x}_1 are just $-\frac{1}{2}\boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{11} \boldsymbol{x}_1$, and hence

$$\boldsymbol{\Lambda}_{1|2}^c = \boldsymbol{\Lambda}_{11} \quad (2.107)$$

The terms which are linear in \boldsymbol{x}_1 are

$$-\frac{1}{2} \boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{12} \boldsymbol{x}_2 - \frac{1}{2} \boldsymbol{x}_2^\top \boldsymbol{\Lambda}_{21} \boldsymbol{x}_1 + \boldsymbol{x}_1^\top \boldsymbol{\eta}_1 = \boldsymbol{x}_1^\top (\boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \boldsymbol{x}_2) \quad (2.108)$$

since $\boldsymbol{\Lambda}_{21}^\top = \boldsymbol{\Lambda}_{12}$. Thus the conditional precision-weighted mean is

$$\boldsymbol{\eta}_{1|2}^c = \boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \boldsymbol{x}_2. \quad (2.109)$$

We will now derive the results for marginalizing in information form. The marginal, $p(\boldsymbol{x}_2)$, can be calculated by integrating the joint, $p(\boldsymbol{x}_1, \boldsymbol{x}_2)$, with respect to \boldsymbol{x}_1 :

$$p(\boldsymbol{x}_2) = \int p(\boldsymbol{x}_1, \boldsymbol{x}_2) d\boldsymbol{x}_1 \quad (2.110)$$

$$\propto \int \exp \left\{ -\frac{1}{2} \boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{11} \boldsymbol{x}_1 - \frac{1}{2} \boldsymbol{x}_2^\top \boldsymbol{\Lambda}_{22} \boldsymbol{x}_2 - \frac{1}{2} \boldsymbol{x}_1^\top \boldsymbol{\Lambda}_{12} \boldsymbol{x}_2 - \frac{1}{2} \boldsymbol{x}_2^\top \boldsymbol{\Lambda}_{21} \boldsymbol{x}_1 + \boldsymbol{x}_1^\top \boldsymbol{\eta}_1 + \boldsymbol{x}_2^\top \boldsymbol{\eta}_2 \right\} d\boldsymbol{x}_1, \quad (2.111)$$

7. This derivation is due to Giles Harper-Donnelly.

where the terms in the exponent have been decomposed into the partitioned structure in Equation (2.104) as in Equation (2.106). Next, collecting all the terms involving \mathbf{x}_1 ,

$$p(\mathbf{x}_2) \propto \exp \left\{ -\frac{1}{2} \mathbf{x}_2^\top \boldsymbol{\Lambda}_{22} \mathbf{x}_2 + \mathbf{x}_2^\top \boldsymbol{\eta}_2 \right\} \int \exp \left\{ -\frac{1}{2} \mathbf{x}_1^\top \boldsymbol{\Lambda}_{11} \mathbf{x}_1 + \mathbf{x}_1^\top (\boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \mathbf{x}_2) \right\} d\mathbf{x}_1, \quad (2.112)$$

we can recognize the integrand as an exponential quadratic form. Therefore the integral is equal to the normalizing constant of a Gaussian with precision, $\boldsymbol{\Lambda}_{11}$, and precision weighted mean, $\boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \mathbf{x}_2$, which is given by the reciprocal of Equation (2.83). Substituting this in to our equation we have,

$$p(\mathbf{x}_2) \propto \exp \left\{ -\frac{1}{2} \mathbf{x}_2^\top \boldsymbol{\Lambda}_{22} \mathbf{x}_2 + \mathbf{x}_2^\top \boldsymbol{\eta}_2 \right\} \exp \left\{ \frac{1}{2} (\boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \mathbf{x}_2)^\top \boldsymbol{\Lambda}_{11}^{-1} (\boldsymbol{\eta}_1 - \boldsymbol{\Lambda}_{12} \mathbf{x}_2) \right\} \quad (2.113)$$

$$\propto \exp \left\{ -\frac{1}{2} \mathbf{x}_2^\top \boldsymbol{\Lambda}_{22} \mathbf{x}_2 + \mathbf{x}_2^\top \boldsymbol{\eta}_2 + \frac{1}{2} \mathbf{x}_2^\top \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\Lambda}_{12} \mathbf{x}_2 - \mathbf{x}_2^\top \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\eta}_1 \right\} \quad (2.114)$$

$$= \exp \left\{ -\frac{1}{2} \mathbf{x}_2^\top (\boldsymbol{\Lambda}_{22} - \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\Lambda}_{12}) \mathbf{x}_2 + \mathbf{x}_2^\top (\boldsymbol{\eta}_2 - \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\eta}_1) \right\}, \quad (2.115)$$

which we now recognise as an exponential quadratic form in \mathbf{x}_2 . Extract the quadratic terms to get the marginal precision,

$$\boldsymbol{\Lambda}_{22}^m = \boldsymbol{\Lambda}_{22} - \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\Lambda}_{12}, \quad (2.116)$$

and the linear terms to get the marginal precision-weighted mean,

$$\boldsymbol{\eta}_2^m = \boldsymbol{\eta}_2 - \boldsymbol{\Lambda}_{21} \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\eta}_1. \quad (2.117)$$

2.3.2 Linear Gaussian systems

Consider two random vectors $\mathbf{y} \in \mathbb{R}^D$ and $\mathbf{z} \in \mathbb{R}^L$, which are jointly Gaussian with the following joint distribution:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z} | \tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}) \quad (2.118)$$

$$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{y} | \mathbf{W}\mathbf{z} + \mathbf{b}, \boldsymbol{\Omega}) \quad (2.119)$$

where \mathbf{W} is a matrix of size $D \times L$. This is an example of a **linear Gaussian system**.

2.3.2.1 Joint distribution

The corresponding joint distribution, $p(\mathbf{z}, \mathbf{y}) = p(\mathbf{z})p(\mathbf{y}|\mathbf{z})$, is itself a $D + L$ dimensional Gaussian, with mean and covariance given by the following (this result can be obtained by moment matching):

$$p(\mathbf{z}, \mathbf{y}) = \mathcal{N}(\mathbf{z}, \mathbf{y} | \tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}) \quad (2.120a)$$

$$\tilde{\boldsymbol{\mu}} \triangleq \begin{pmatrix} \tilde{\boldsymbol{\mu}} \\ \mathbf{m} \end{pmatrix} \triangleq \begin{pmatrix} \tilde{\boldsymbol{\mu}} \\ \mathbf{W} \tilde{\boldsymbol{\mu}} + \mathbf{b} \end{pmatrix} \quad (2.120b)$$

$$\tilde{\boldsymbol{\Sigma}} \triangleq \begin{pmatrix} \tilde{\boldsymbol{\Sigma}} & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{S} \end{pmatrix} \triangleq \begin{pmatrix} \tilde{\boldsymbol{\Sigma}} & \tilde{\boldsymbol{\Sigma}} \mathbf{W}^\top \\ \mathbf{W} \tilde{\boldsymbol{\Sigma}} & \mathbf{W} \tilde{\boldsymbol{\Sigma}} \mathbf{W}^\top + \boldsymbol{\Omega} \end{pmatrix} \quad (2.120c)$$

See Algorithm 8.1 on page 363 for some pseudocode to compute this joint distribution.

2.3.2.2 Posterior distribution (Bayes' rule for Gaussians)

Now we consider computing the posterior $p(\mathbf{z}|\mathbf{y})$ from a linear Gaussian system. Using Equation (2.78) for conditioning a joint Gaussian, we find that the posterior is given by

$$p(\mathbf{z}|\mathbf{y}) = \mathcal{N}(\mathbf{z} | \hat{\boldsymbol{\mu}}, \hat{\Sigma}) \quad (2.121a)$$

$$\hat{\boldsymbol{\mu}} = \check{\boldsymbol{\mu}} + \check{\Sigma} \mathbf{W}^T (\Omega + \mathbf{W} \check{\Sigma} \mathbf{W}^T)^{-1} (\mathbf{y} - (\mathbf{W} \check{\boldsymbol{\mu}} + \mathbf{b})) \quad (2.121b)$$

$$\hat{\Sigma} = \check{\Sigma} - \check{\Sigma} \mathbf{W}^T (\Omega + \mathbf{W} \check{\Sigma} \mathbf{W}^T)^{-1} \mathbf{W} \check{\Sigma} \quad (2.121c)$$

This is known as **Bayes' rule for Gaussians**. We see that if the prior $p(\mathbf{z})$ is Gaussian, and the likelihood $p(\mathbf{y}|\mathbf{z})$ is Gaussian, then the posterior $p(\mathbf{z}|\mathbf{y})$ is also Gaussian. We therefore say that the Gaussian prior is a **conjugate prior** for the Gaussian likelihood, since the posterior distribution has the same type as the prior. (In other words, Gaussians are closed under Bayesian updating.)

We can simplify these equations by defining $\mathbf{S} = \mathbf{W} \check{\Sigma} \mathbf{W}^T + \Omega$, $\mathbf{C} = \check{\Sigma} \mathbf{W}^T$, and $\mathbf{m} = \mathbf{W} \check{\boldsymbol{\mu}} + \mathbf{b}$, as in Equation (2.120). We also define the **Kalman gain matrix**:⁸

$$\mathbf{K} = \mathbf{C} \mathbf{S}^{-1} \quad (2.122)$$

From this, we get the posterior

$$\hat{\boldsymbol{\mu}} = \check{\boldsymbol{\mu}} + \mathbf{K}(\mathbf{y} - \mathbf{m}) \quad (2.123)$$

$$\hat{\Sigma} = \check{\Sigma} - \mathbf{K} \mathbf{C}^T \quad (2.124)$$

Note that

$$\mathbf{K} \mathbf{S} \mathbf{K}^T = \mathbf{C} \mathbf{S}^{-1} \mathbf{S} \mathbf{S}^{-T} \mathbf{C}^T = \mathbf{C} \mathbf{S}^{-1} \mathbf{C}^T = \mathbf{K} \mathbf{C}^T \quad (2.125)$$

and hence we can also write the posterior covariance as

$$\hat{\Sigma} = \check{\Sigma} - \mathbf{K} \mathbf{S} \mathbf{K}^T \quad (2.126)$$

Using the matrix inversion lemma from Equation (2.93), we can also rewrite the posterior in the following form [Bis06, p93], which takes $O(L^3)$ time instead of $O(D^3)$ time:

$$\hat{\Sigma} = (\check{\Sigma}^{-1} + \mathbf{W}^T \Omega^{-1} \mathbf{W})^{-1} \quad (2.127)$$

$$\hat{\boldsymbol{\mu}} = \hat{\Sigma} [\mathbf{W}^T \Omega^{-1} (\mathbf{y} - \mathbf{b}) + \check{\Sigma}^{-1} \check{\boldsymbol{\mu}}] \quad (2.128)$$

Finally, note that the corresponding normalization constant for the posterior is just the marginal on \mathbf{y} evaluated at the observed value:

$$\begin{aligned} p(\mathbf{y}) &= \int \mathcal{N}(\mathbf{z} | \check{\boldsymbol{\mu}}, \check{\Sigma}) \mathcal{N}(\mathbf{y} | \mathbf{W}\mathbf{z} + \mathbf{b}, \Omega) d\mathbf{z} \\ &= \mathcal{N}(\mathbf{y} | \mathbf{W} \check{\boldsymbol{\mu}} + \mathbf{b}, \Omega + \mathbf{W} \check{\Sigma} \mathbf{W}^T) = \mathcal{N}(\mathbf{y} | \mathbf{m}, \mathbf{S}) \end{aligned} \quad (2.129)$$

From this, we can easily compute the log marginal likelihood. We summarize all these equations in Algorithm 8.1.

8. The name comes from the Kalman filter algorithm, which we discuss in Section 8.2.2.

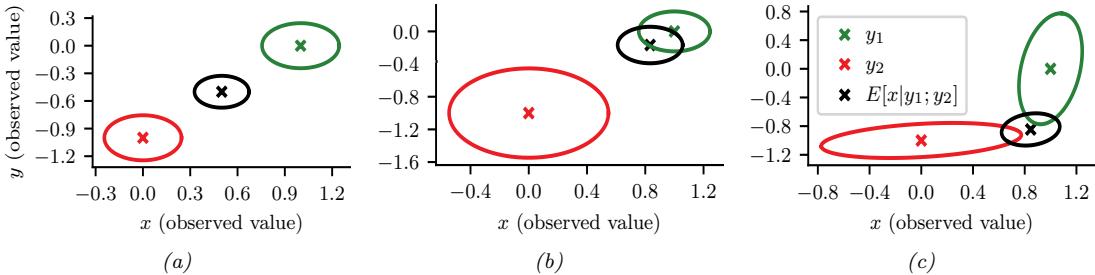


Figure 2.10: We observe $\mathbf{x} = (0, -1)$ (red cross) and $\mathbf{y} = (1, 0)$ (green cross) and estimate $\mathbb{E}[z|\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}]$ (black cross). (a) Equally reliable sensors, so the posterior mean estimate is in between the two circles. (b) Sensor 2 is more reliable, so the estimate shifts more towards the green circle. (c) Sensor 1 is more reliable in the vertical direction, Sensor 2 is more reliable in the horizontal direction. The estimate is an appropriate combination of the two measurements. Generated by `sensor_fusion_2d.ipynb`.

2.3.2.3 Example: Sensor fusion with known measurement noise

Suppose we have an unknown quantity of interest, $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$, from which we get two noisy measurements, $\mathbf{x} \sim \mathcal{N}(\mathbf{z}, \boldsymbol{\Sigma}_x)$ and $\mathbf{y} \sim \mathcal{N}(\mathbf{z}, \boldsymbol{\Sigma}_y)$. Pictorially, we can represent this example as $\mathbf{x} \leftarrow \mathbf{z} \rightarrow \mathbf{y}$. This is an example of a linear Gaussian system. Our goal is to combine the evidence together, to compute $p(\mathbf{z}|\mathbf{x}, \mathbf{y}; \boldsymbol{\theta})$. This is known as **sensor fusion**. (In this section, we assume $\boldsymbol{\theta} = (\boldsymbol{\Sigma}_x, \boldsymbol{\Sigma}_y)$ is known. See [Supplementary](#) Section 2.1.2 for the general case.)

We can combine \mathbf{x} and \mathbf{y} into a single vector \mathbf{v} , so the model can be represented as $\mathbf{z} \rightarrow \mathbf{v}$, where $p(\mathbf{v}|\mathbf{z}) = \mathcal{N}(\mathbf{v}|\mathbf{W}\mathbf{z}, \Sigma_v)$, where $\mathbf{W} = [\mathbf{I}; \mathbf{I}]$ and $\Sigma_v = [\Sigma_x; \mathbf{0}; \mathbf{0}; \Sigma_y]$ are block-structured matrices. We can then apply Bayes' rule for Gaussians (Section 2.3.2.2) to compute $p(\mathbf{z}|\mathbf{v})$.

Figure 2.10(a) gives a 2d example, where we set $\Sigma_x = \Sigma_y = 0.01\mathbf{I}_2$, so both sensors are equally reliable. In this case, the posterior mean is halfway between the two observations, \mathbf{x} and \mathbf{y} . In Figure 2.10(b), we set $\Sigma_x = 0.05\mathbf{I}_2$ and $\Sigma_y = 0.01\mathbf{I}_2$, so sensor 2 is more reliable than sensor 1. In this case, the posterior mean is closer to \mathbf{y} . In Figure 2.10(c), we set

$$\boldsymbol{\Sigma}_x = 0.01 \begin{pmatrix} 10 & 1 \\ 1 & 1 \end{pmatrix}, \quad \boldsymbol{\Sigma}_y = 0.01 \begin{pmatrix} 1 & 1 \\ 1 & 10 \end{pmatrix} \quad (2.130)$$

so sensor 1 is more reliable in the second component (vertical direction), and sensor 2 is more reliable in the first component (horizontal direction). In this case, the posterior mean is vertically closer to \mathbf{x} and horizontally closer to \mathbf{y} .

2.3.3 A general calculus for linear Gaussian systems

In this section, we discuss a general method for performing inference in linear Gaussian systems. The key is to define joint distributions over the relevant variables in terms of a **potential function**, represented in information form. We can then easily derive rules for marginalizing potentials, multiplying and dividing potentials, and conditioning them on observations. Once we have defined these operations, we can use them inside of the belief propagation algorithm (Section 9.3) or junction tree algorithm ([Supplementary](#) Section 9.2) to compute quantities of interest. We give the details on how to perform these operations below; our presentation is based on [[Lau92](#); [Mur02](#)].

2.3.3.1 Moment and canonical parameterization

We can represent a Gaussian distribution in moment form or in canonical (information) form. In moment form we have

$$\phi(\mathbf{x}; p, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = p \times \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \quad (2.131)$$

where $p = (2\pi)^{-n/2} |\boldsymbol{\Sigma}|^{-\frac{1}{2}}$ is the normalizing constant that ensures $\int_{\mathbf{x}} \phi(\mathbf{x}; p, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = 1$. (n is the dimensionality of \mathbf{x} .) Expanding out the quadratic form and collecting terms we get the canonical form:

$$\phi(\mathbf{x}; g, \mathbf{h}, \mathbf{K}) = \exp\left(g + \mathbf{x}^\top \mathbf{h} - \frac{1}{2} \mathbf{x}^\top \mathbf{K} \mathbf{x}\right) = \exp\left(g + \sum_i h_i x_i - \frac{1}{2} \sum_i \sum_k K_{ij} x_i x_j\right) \quad (2.132)$$

where

$$\mathbf{K} = \boldsymbol{\Sigma}^{-1} \quad (2.133)$$

$$\mathbf{h} = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \quad (2.134)$$

$$g = \log p - \frac{1}{2} \boldsymbol{\mu}^\top \mathbf{K} \boldsymbol{\mu} \quad (2.135)$$

\mathbf{K} is often called the precision matrix.

Note that potentials need not be probability distributions, and need not be normalizable (integrate to 1). We keep track of the constant terms (p or g) so we can compute the likelihood of the evidence.

2.3.3.2 Multiplication and division

We can define multiplication and division in the Gaussian case by using canonical forms, as follows. To multiply $\phi_1(x_1, \dots, x_k; g_1, \mathbf{h}_1, \mathbf{K}_1)$ by $\phi_2(x_{k+1}, \dots, x_n; g_2, \mathbf{h}_2, \mathbf{K}_2)$, we extend them both to the same domain x_1, \dots, x_n by adding zeros to the appropriate dimensions, and then computing

$$(g_1, \mathbf{h}_1, \mathbf{K}_1) * (g_2, \mathbf{h}_2, \mathbf{K}_2) = (g_1 + g_2, \mathbf{h}_1 + \mathbf{h}_2, \mathbf{K}_1 + \mathbf{K}_2) \quad (2.136)$$

Division is defined as follows:

$$(g_1, \mathbf{h}_1, \mathbf{K}_1) / (g_2, \mathbf{h}_2, \mathbf{K}_2) = (g_1 - g_2, \mathbf{h}_1 - \mathbf{h}_2, \mathbf{K}_1 - \mathbf{K}_2) \quad (2.137)$$

2.3.3.3 Marginalization

Let ϕ_W be a potential over a set W of variables. We can compute the potential over a subset $V \subset W$ of variables by marginalizing, denoted $\phi_V = \sum_{W \setminus V} \phi_W$. Let

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}, \quad \mathbf{h} = \begin{pmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \end{pmatrix}, \quad \mathbf{K} = \begin{pmatrix} \mathbf{K}_{11} & \mathbf{K}_{12} \\ \mathbf{K}_{21} & \mathbf{K}_{22} \end{pmatrix}, \quad (2.138)$$

with \mathbf{x}_1 having dimension n_1 and \mathbf{x}_2 having dimension n_2 . It can be shown that

$$\int_{\mathbf{x}_1} \phi(\mathbf{x}_1, \mathbf{x}_2; g, \mathbf{h}, \mathbf{K}) = \phi(\mathbf{x}_2; \hat{g}, \hat{\mathbf{h}}, \hat{\mathbf{K}}) \quad (2.139)$$

where

$$\hat{g} = g + \frac{1}{2} (n_1 \log(2\pi) - \log |\mathbf{K}_{11}| + \mathbf{h}_1^\top \mathbf{K}_{11}^{-1} \mathbf{h}_1) \quad (2.140)$$

$$\hat{\mathbf{h}} = \mathbf{h}_2 - \mathbf{K}_{21} \mathbf{K}_{11}^{-1} \mathbf{h}_1 \quad (2.141)$$

$$\hat{\mathbf{K}} = \mathbf{K}_{22} - \mathbf{K}_{21} \mathbf{K}_{11}^{-1} \mathbf{K}_{12} \quad (2.142)$$

2.3.3.4 Conditioning on evidence

Consider a potential defined on (\mathbf{x}, \mathbf{y}) . Suppose we observe the value \mathbf{y} . The new potential is given by the following reduced dimensionality object:

$$\phi^*(\mathbf{x}) = \exp \left[g + (\mathbf{x}^T \quad \mathbf{y}^T) \begin{pmatrix} \mathbf{h}_X \\ \mathbf{h}_Y \end{pmatrix} - \frac{1}{2} (\mathbf{x}^T \quad \mathbf{y}^T) \begin{pmatrix} \mathbf{K}_{XX} & \mathbf{K}_{XY} \\ \mathbf{K}_{YX} & \mathbf{K}_{YY} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \right] \quad (2.143)$$

$$= \exp \left[\left(g + \mathbf{h}_Y^T \mathbf{y} - \frac{1}{2} \mathbf{y}^T \mathbf{K}_{YY} \mathbf{y} \right) + \mathbf{x}^T (\mathbf{h}_X - \mathbf{K}_{XY} \mathbf{y}) - \frac{1}{2} \mathbf{x}^T \mathbf{K}_{XX} \mathbf{x} \right] \quad (2.144)$$

This generalizes the corresponding equation in [Lau92] to the vector-valued case.

2.3.3.5 Converting a linear-Gaussian CPD to a canonical potential

Finally we discuss how to create the initial potentials, assuming we start with a directed Gaussian graphical model. In particular, consider a node with a linear-Gaussian conditional probability distribution (CPD):

$$p(\mathbf{x}|\mathbf{u}) = c \exp \left[-\frac{1}{2} ((\mathbf{x} - \boldsymbol{\mu} - \mathbf{B}^T \mathbf{u})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu} - \mathbf{B}^T \mathbf{u})) \right] \quad (2.145)$$

$$= \exp \left[-\frac{1}{2} (\mathbf{x} - \mathbf{u}) \begin{pmatrix} \boldsymbol{\Sigma}^{-1} & -\boldsymbol{\Sigma}^{-1} \mathbf{B}^T \\ -\mathbf{B} \boldsymbol{\Sigma}^{-1} & \mathbf{B} \boldsymbol{\Sigma}^{-1} \mathbf{B}^T \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{u} \end{pmatrix} \right] \quad (2.146)$$

$$+ (\mathbf{x} - \mathbf{u}) \begin{pmatrix} \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \\ -\mathbf{B} \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \end{pmatrix} - \frac{1}{2} \boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} + \log c \quad (2.147)$$

where $c = (2\pi)^{-n/2} |\boldsymbol{\Sigma}|^{-\frac{1}{2}}$. Hence we set the canonical parameters to

$$g = -\frac{1}{2} \boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} - \frac{n}{2} \log(2\pi) - \frac{1}{2} \log |\boldsymbol{\Sigma}| \quad (2.148)$$

$$\mathbf{h} = \begin{pmatrix} \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \\ -\mathbf{B} \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \end{pmatrix} \quad (2.149)$$

$$\mathbf{K} = \begin{pmatrix} \boldsymbol{\Sigma}^{-1} & -\boldsymbol{\Sigma}^{-1} \mathbf{B}^T \\ -\mathbf{B} \boldsymbol{\Sigma}^{-1} & \mathbf{B} \boldsymbol{\Sigma}^{-1} \mathbf{B}^T \end{pmatrix} = \begin{pmatrix} \mathbf{I} \\ -\mathbf{B} \end{pmatrix} \boldsymbol{\Sigma}^{-1} (\mathbf{I} \quad -\mathbf{B}) \quad (2.150)$$

In the special case that x is a scalar, the corresponding result can be found in [Lau92]. In particular

we have $\Sigma^{-1} = 1/\sigma^2$, $B = b$ and $n = 1$, so the above becomes

$$g = \frac{-\mu^2}{2\sigma^2} - \frac{1}{2} \log(2\pi\sigma^2) \quad (2.151)$$

$$\mathbf{h} = \frac{\mu}{\sigma^2} \begin{pmatrix} 1 \\ -b \end{pmatrix} \quad (2.152)$$

$$\mathbf{K} = \frac{1}{\sigma} \begin{pmatrix} 1 & -b^T \\ -b & bb^T \end{pmatrix}. \quad (2.153)$$

2.3.3.6 Example: Product of Gaussians

As an application of the above results, we can derive the (unnormalized) product of two Gaussians, as follows (see also [Kaa12, Sec 8.1.8]):

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \times \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \propto \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3) \quad (2.154)$$

where

$$\boldsymbol{\Sigma}_3 = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1} \quad (2.155)$$

$$\boldsymbol{\mu}_3 = \boldsymbol{\Sigma}_3(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2) \quad (2.156)$$

We see that the posterior precision is a sum of the individual precisions, and the posterior mean is a precision-weighted combination of the individual means. We can also rewrite the result in the following way, which only requires one matrix inversion:

$$\boldsymbol{\Sigma}_3 = \boldsymbol{\Sigma}_1(\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2)^{-1}\boldsymbol{\Sigma}_2 \quad (2.157)$$

$$\boldsymbol{\mu}_3 = \boldsymbol{\Sigma}_2(\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2)^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_1(\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2)^{-1}\boldsymbol{\mu}_2 \quad (2.158)$$

In the scalar case, this becomes

$$\mathcal{N}(x|\mu_1, \sigma_1^2)\mathcal{N}(x|\mu_2, \sigma_2^2) \propto \mathcal{N}\left(x \mid \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2}, \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}\right) \quad (2.159)$$

2.4 The exponential family

In this section, we define the **exponential family**, which includes many common probability distributions. The exponential family plays a crucial role in statistics and machine learning, for various reasons, including the following:

- The exponential family is the unique family of distributions that has maximum entropy (and hence makes the least set of assumptions) subject to some user-chosen constraints, as discussed in Section 2.4.7.
- The exponential family is at the core of GLMs, as discussed in Section 15.1.
- The exponential family is at the core of variational inference, as discussed in Chapter 10.

- Under certain regularity conditions, the exponential family is the only family of distributions with finite-sized sufficient statistics, as discussed in Section 2.4.5.
- All members of the exponential family have a **conjugate prior** [DY79], which simplifies Bayesian inference of the parameters, as discussed in Section 3.4.

2.4.1 Definition

Consider a family of probability distributions parameterized by $\boldsymbol{\eta} \in \mathbb{R}^K$ with fixed support over $\mathcal{X}^D \subseteq \mathbb{R}^D$. We say that the distribution $p(\mathbf{x}|\boldsymbol{\eta})$ is in the **exponential family** if its density can be written in the following way:

$$p(\mathbf{x}|\boldsymbol{\eta}) \triangleq \frac{1}{Z(\boldsymbol{\eta})} h(\mathbf{x}) \exp[\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x})] = h(\mathbf{x}) \exp[\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})] \quad (2.160)$$

where $h(\mathbf{x})$ is a scaling constant (also known as the **base measure**, often 1), $\mathcal{T}(\mathbf{x}) \in \mathbb{R}^K$ are the **sufficient statistics**, $\boldsymbol{\eta}$ are the **natural parameters** or **canonical parameters**, $Z(\boldsymbol{\eta})$ is a normalization constant known as the **partition function**, and $A(\boldsymbol{\eta}) = \log Z(\boldsymbol{\eta})$ is the **log partition function**. In Section 2.4.3, we show that A is a convex function over the convex set $\Omega \triangleq \{\boldsymbol{\eta} \in \mathbb{R}^K : A(\boldsymbol{\eta}) < \infty\}$.

It is convenient if the natural parameters are independent of each other. Formally, we say that an exponential family is **minimal** if there is no $\boldsymbol{\eta} \in \mathbb{R}^K \setminus \{0\}$ such that $\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x}) = 0$. This last condition can be violated in the case of multinomial distributions, because of the sum to one constraint on the parameters; however, it is easy to reparameterize the distribution using $K - 1$ independent parameters, as we show below.

Equation (2.160) can be generalized by defining $\boldsymbol{\eta} = f(\boldsymbol{\phi})$, where $\boldsymbol{\phi}$ is some other, possibly smaller, set of parameters. In this case, the distribution has the form

$$p(\mathbf{x}|\boldsymbol{\phi}) = h(\mathbf{x}) \exp[f(\boldsymbol{\phi})^\top \mathcal{T}(\mathbf{x}) - A(f(\boldsymbol{\phi}))] \quad (2.161)$$

If the mapping from $\boldsymbol{\phi}$ to $\boldsymbol{\eta}$ is nonlinear, we call this a **curved exponential family**. If $\boldsymbol{\eta} = f(\boldsymbol{\phi}) = \boldsymbol{\phi}$, the model is said to be in **canonical form**. If, in addition, $\mathcal{T}(\mathbf{x}) = \mathbf{x}$, we say this is a **natural exponential family** or **NEF**. In this case, it can be written as

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x}) \exp[\boldsymbol{\eta}^\top \mathbf{x} - A(\boldsymbol{\eta})] \quad (2.162)$$

We define the **moment parameters** as the mean of the sufficient statistics vector:

$$\mathbf{m} = \mathbb{E}[\mathcal{T}(\mathbf{x})] \quad (2.163)$$

We will see some examples below.

2.4.2 Examples

In this section, we consider some common examples of distributions in the exponential family. Each corresponds to a different way of defining $h(\mathbf{x})$ and $\mathcal{T}(\mathbf{x})$ (since Z and hence A are derived from knowing h and \mathcal{T}).

2.4.2.1 Bernoulli distribution

The Bernoulli distribution can be written in exponential family form as follows:

$$\text{Ber}(x|\mu) = \mu^x(1-\mu)^{1-x} \quad (2.164)$$

$$= \exp[x \log(\mu) + (1-x) \log(1-\mu)] \quad (2.165)$$

$$= \exp[\mathcal{T}(x)^\top \boldsymbol{\eta}] \quad (2.166)$$

where $\mathcal{T}(x) = [\mathbb{I}(x=1), \mathbb{I}(x=0)]$, $\boldsymbol{\eta} = [\log(\mu), \log(1-\mu)]$, and μ is the mean parameter. However, this is an **over-complete representation** since there is a linear dependence between the features. We can see this as follows:

$$\mathbf{1}^\top \mathcal{T}(x) = \mathbb{I}(x=0) + \mathbb{I}(x=1) = 1 \quad (2.167)$$

If the representation is overcomplete, $\boldsymbol{\eta}$ is not uniquely identifiable. It is common to use a **minimal representation**, which means there is a unique $\boldsymbol{\eta}$ associated with the distribution. In this case, we can just define

$$\text{Ber}(x|\mu) = \exp \left[x \log \left(\frac{\mu}{1-\mu} \right) + \log(1-\mu) \right] \quad (2.168)$$

We can put this into exponential family form by defining

$$\eta = \log \left(\frac{\mu}{1-\mu} \right) \quad (2.169)$$

$$\mathcal{T}(x) = x \quad (2.170)$$

$$A(\eta) = -\log(1-\mu) = \log(1+e^\eta) \quad (2.171)$$

$$h(x) = 1 \quad (2.172)$$

We can recover the mean parameter μ from the canonical parameter η using

$$\mu = \sigma(\eta) = \frac{1}{1+e^{-\eta}} \quad (2.173)$$

which we recognize as the logistic (sigmoid) function.

2.4.2.2 Categorical distribution

We can represent the discrete distribution with K categories as follows (where $x_k = \mathbb{I}(x = k)$):

$$\text{Cat}(x|\boldsymbol{\mu}) = \prod_{k=1}^K \mu_k^{x_k} = \exp \left[\sum_{k=1}^K x_k \log \mu_k \right] \quad (2.174)$$

$$= \exp \left[\sum_{k=1}^{K-1} x_k \log \mu_k + \left(1 - \sum_{k=1}^{K-1} x_k \right) \log \left(1 - \sum_{k=1}^{K-1} \mu_k \right) \right] \quad (2.175)$$

$$= \exp \left[\sum_{k=1}^{K-1} x_k \log \left(\frac{\mu_k}{1 - \sum_{j=1}^{K-1} \mu_j} \right) + \log \left(1 - \sum_{k=1}^{K-1} \mu_k \right) \right] \quad (2.176)$$

$$= \exp \left[\sum_{k=1}^{K-1} x_k \log \left(\frac{\mu_k}{\mu_K} \right) + \log \mu_K \right] \quad (2.177)$$

where $\mu_K = 1 - \sum_{k=1}^{K-1} \mu_k$. We can write this in exponential family form as follows:

$$\text{Cat}(x|\boldsymbol{\eta}) = \exp(\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})) \quad (2.178)$$

$$\boldsymbol{\eta} = [\log \frac{\mu_1}{\mu_K}, \dots, \log \frac{\mu_{K-1}}{\mu_K}] \quad (2.179)$$

$$A(\boldsymbol{\eta}) = -\log(\mu_K) \quad (2.180)$$

$$\mathcal{T}(x) = [\mathbb{I}(x=1), \dots, \mathbb{I}(x=K-1)] \quad (2.181)$$

$$h(x) = 1 \quad (2.182)$$

We can recover the mean parameters from the canonical parameters using

$$\mu_k = \frac{e^{\eta_k}}{1 + \sum_{j=1}^{K-1} e^{\eta_j}} \quad (2.183)$$

If we define $\eta_K = 0$, we can rewrite this as follows:

$$\mu_k = \frac{e^{\eta_k}}{\sum_{j=1}^K e^{\eta_j}} \quad (2.184)$$

for $k = 1 : K$. Hence $\boldsymbol{\mu} = \text{softmax}(\boldsymbol{\eta})$, where softmax is the softmax or multinomial logit function in Equation (15.136). From this, we find

$$\mu_K = 1 - \frac{\sum_{k=1}^{K-1} e^{\eta_k}}{1 + \sum_{k=1}^{K-1} e^{\eta_k}} = \frac{1}{1 + \sum_{k=1}^{K-1} e^{\eta_k}} \quad (2.185)$$

and hence

$$A(\boldsymbol{\eta}) = -\log(\mu_K) = \log \left(\sum_{k=1}^K e^{\eta_k} \right) \quad (2.186)$$

2.4.2.3 Univariate Gaussian

The univariate Gaussian is usually written as follows:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{\frac{1}{2}}} \exp\left[-\frac{1}{2\sigma^2}(x - \mu)^2\right] \quad (2.187)$$

$$= \frac{1}{(2\pi)^{\frac{1}{2}}} \exp\left[\frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}x^2 - \frac{1}{2\sigma^2}\mu^2 - \log\sigma\right] \quad (2.188)$$

We can put this in exponential family form by defining

$$\boldsymbol{\eta} = \begin{pmatrix} \mu/\sigma^2 \\ -\frac{1}{2\sigma^2} \end{pmatrix} \quad (2.189)$$

$$\mathcal{T}(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix} \quad (2.190)$$

$$A(\boldsymbol{\eta}) = \frac{\mu^2}{2\sigma^2} + \log\sigma = \frac{-\eta_1^2}{4\eta_2} - \frac{1}{2}\log(-2\eta_2) \quad (2.191)$$

$$h(x) = \frac{1}{\sqrt{2\pi}} \quad (2.192)$$

The moment parameters are

$$\mathbf{m} = [\mu, \mu^2 + \sigma^2] \quad (2.193)$$

2.4.2.4 Univariate Gaussian with fixed variance

If we fix $\sigma^2 = 1$, we can write the Gaussian as a natural exponential family, by defining

$$\eta = \mu \quad (2.194)$$

$$\mathcal{T}(x) = x \quad (2.195)$$

$$A(\mu) = \frac{\mu^2}{2\sigma^2} + \log\sigma = \frac{\mu^2}{2} \quad (2.196)$$

$$h(x) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right] = \mathcal{N}(x|0, 1) \quad (2.197)$$

Note that this in example, the base measure $h(x)$ is not constant.

2.4.2.5 Multivariate Gaussian

It is common to parameterize the multivariate normal (MVN) in terms of the mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$. The corresponding pdf is given by

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \mathbf{x} + \mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} - \frac{1}{2}\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}\right) \quad (2.198)$$

$$= c \exp\left(\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} - \frac{1}{2}\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \mathbf{x}\right) \quad (2.199)$$

$$c \triangleq \frac{\exp(-\frac{1}{2}\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu})}{(2\pi)^{D/2}\sqrt{\det(\boldsymbol{\Sigma})}} \quad (2.200)$$

However, we can also represent the Gaussian using **canonical parameters** or **natural parameters**, also called the **information form**:

$$\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} \quad (2.201)$$

$$\boldsymbol{\xi} = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} \quad (2.202)$$

$$\mathcal{N}_c(\mathbf{x}|\boldsymbol{\xi}, \boldsymbol{\Lambda}) \triangleq c' \exp\left(\mathbf{x}^\top \boldsymbol{\xi} - \frac{1}{2}\mathbf{x}^\top \boldsymbol{\Lambda} \mathbf{x}\right) \quad (2.203)$$

$$c' = \frac{\exp(-\frac{1}{2}\boldsymbol{\xi}^\top \boldsymbol{\Lambda}^{-1} \boldsymbol{\xi})}{(2\pi)^{D/2}\sqrt{\det(\boldsymbol{\Lambda}^{-1})}} \quad (2.204)$$

where we use the notation $\mathcal{N}_c()$ to distinguish it from the standard parameterization $\mathcal{N}()$. Here $\boldsymbol{\Lambda}$ is called the **precision matrix** and $\boldsymbol{\xi}$ is the precision-weighted mean vector.

We can convert this to exponential family notation as follows:

$$\mathcal{N}_c(\mathbf{x}|\boldsymbol{\xi}, \boldsymbol{\Lambda}) = \underbrace{(2\pi)^{-D/2}}_{h(\mathbf{x})} \underbrace{\exp\left[\frac{1}{2}\log|\boldsymbol{\Lambda}| - \frac{1}{2}\boldsymbol{\xi}^\top \boldsymbol{\Lambda}^{-1} \boldsymbol{\xi}\right]}_{g(\boldsymbol{\eta})} \exp\left[-\frac{1}{2}\mathbf{x}^\top \boldsymbol{\Lambda} \mathbf{x} + \mathbf{x}^\top \boldsymbol{\xi}\right] \quad (2.205)$$

$$= h(\mathbf{x})g(\boldsymbol{\eta}) \exp\left[-\frac{1}{2}\mathbf{x}^\top \boldsymbol{\Lambda} \mathbf{x} + \mathbf{x}^\top \boldsymbol{\xi}\right] \quad (2.206)$$

$$= h(\mathbf{x})g(\boldsymbol{\eta}) \exp\left[-\frac{1}{2}\left(\sum_{ij} x_i x_j \Lambda_{ij}\right) + \mathbf{x}^\top \boldsymbol{\xi}\right] \quad (2.207)$$

$$= h(\mathbf{x})g(\boldsymbol{\eta}) \exp\left[-\frac{1}{2}\text{vec}(\boldsymbol{\Lambda})^\top \text{vec}(\mathbf{x}\mathbf{x}^\top) + \mathbf{x}^\top \boldsymbol{\xi}\right] \quad (2.208)$$

$$= h(\mathbf{x}) \exp[\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})] \quad (2.209)$$

where

$$h(\mathbf{x}) = (2\pi)^{-D/2} \quad (2.210)$$

$$\boldsymbol{\eta} = [\boldsymbol{\xi}; -\frac{1}{2}\text{vec}(\boldsymbol{\Lambda})] = [\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}; -\frac{1}{2}\text{vec}(\boldsymbol{\Sigma}^{-1})] \quad (2.211)$$

$$\mathcal{T}(\mathbf{x}) = [\mathbf{x}; \text{vec}(\mathbf{x}\mathbf{x}^\top)] \quad (2.212)$$

$$A(\boldsymbol{\eta}) = -\log g(\boldsymbol{\eta}) = -\frac{1}{2}\log|\boldsymbol{\Lambda}| + \frac{1}{2}\boldsymbol{\xi}^\top\boldsymbol{\Lambda}^{-1}\boldsymbol{\xi} \quad (2.213)$$

From this, we see that the mean (moment) parameters are given by

$$\mathbf{m} = \mathbb{E}[\mathcal{T}(\mathbf{x})] = [\boldsymbol{\mu}; \boldsymbol{\mu}\boldsymbol{\mu}^\top + \boldsymbol{\Sigma}] \quad (2.214)$$

(Note that the above is not a minimal representation, since $\boldsymbol{\Lambda}$ is a symmetric matrix. We can convert to minimal form by working with the upper or lower half of each matrix.)

2.4.2.6 Non-examples

Not all distributions of interest belong to the exponential family. For example, the Student distribution (Section 2.2.3) does not belong, since its pdf (Equation (2.30)) does not have the required form. (However, there is a generalization, known as the ϕ -exponential family [Nau04; Tsa88] which does include the Student distribution.)

As a more subtle example, consider the uniform distribution, $Y \sim \text{Unif}(\theta_1, \theta_2)$. The pdf has the form

$$p(y|\boldsymbol{\theta}) = \frac{1}{\theta_2 - \theta_1} \mathbb{I}(\theta_1 \leq y \leq \theta_2) \quad (2.215)$$

It is tempting to think this is in the exponential family, with $h(y) = 1$, $\mathcal{T}(y) = \mathbf{0}$, and $Z(\boldsymbol{\theta}) = \theta_2 - \theta_1$. However, the support of this distribution (i.e., the set of values $\mathcal{Y} = \{y : p(y) > 0\}$) depends on the parameters $\boldsymbol{\theta}$, which violates an assumption of the exponential family.

2.4.3 Log partition function is cumulant generating function

The first and second **cumulants** of a distribution are its mean $\mathbb{E}[X]$ and variance $\mathbb{V}[X]$, whereas the first and second moments are $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$. We can also compute higher order cumulants (and moments). An important property of the exponential family is that derivatives of the log partition function can be used to generate all the **cumulants** of the sufficient statistics. In particular, the first and second cumulants are given by

$$\nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) = \mathbb{E}[\mathcal{T}(\mathbf{x})] \quad (2.216)$$

$$\nabla_{\boldsymbol{\eta}}^2 A(\boldsymbol{\eta}) = \text{Cov}[\mathcal{T}(\mathbf{x})] \quad (2.217)$$

We prove this result below.

2.4.3.1 Derivation of the mean

For simplicity, we focus on the 1d case. For the first derivative we have

$$\frac{dA}{d\eta} = \frac{d}{d\eta} \left(\log \int \exp(\eta \mathcal{T}(x)) h(x) dx \right) \quad (2.218)$$

$$= \frac{\frac{d}{d\eta} \int \exp(\eta \mathcal{T}(x)) h(x) dx}{\int \exp(\eta \mathcal{T}(x)) h(x) dx} \quad (2.219)$$

$$= \frac{\int \mathcal{T}(x) \exp(\eta \mathcal{T}(x)) h(x) dx}{\exp(A(\eta))} \quad (2.220)$$

$$= \int \mathcal{T}(x) \exp(\eta \mathcal{T}(x) - A(\eta)) h(x) dx \quad (2.221)$$

$$= \int \mathcal{T}(x) p(x) dx = \mathbb{E}[\mathcal{T}(x)] \quad (2.222)$$

For example, consider the Bernoulli distribution. We have $A(\eta) = \log(1 + e^\eta)$, so the mean is given by

$$\frac{dA}{d\eta} = \frac{e^\eta}{1 + e^\eta} = \frac{1}{1 + e^{-\eta}} = \sigma(\eta) = \mu \quad (2.223)$$

2.4.3.2 Derivation of the variance

For simplicity, we focus on the 1d case. For the second derivative we have

$$\frac{d^2 A}{d\eta^2} = \frac{d}{d\eta} \int \mathcal{T}(x) \exp(\eta \mathcal{T}(x) - A(\eta)) h(x) dx \quad (2.224)$$

$$= \int \mathcal{T}(x) \exp(\eta \mathcal{T}(x) - A(\eta)) h(x) (\mathcal{T}(x) - A'(\eta)) dx \quad (2.225)$$

$$= \int \mathcal{T}(x) p(x) (\mathcal{T}(x) - A'(\eta)) dx \quad (2.226)$$

$$= \int \mathcal{T}^2(x) p(x) dx - A'(\eta) \int \mathcal{T}(x) p(x) dx \quad (2.227)$$

$$= \mathbb{E}[\mathcal{T}^2(X)] - \mathbb{E}[\mathcal{T}(x)]^2 = \mathbb{V}[\mathcal{T}(x)] \quad (2.228)$$

where we used the fact that $A'(\eta) = \frac{dA}{d\eta} = \mathbb{E}[\mathcal{T}(x)]$. For example, for the Bernoulli distribution we have

$$\frac{d^2 A}{d\eta^2} = \frac{d}{d\eta} (1 + e^{-\eta})^{-1} = (1 + e^{-\eta})^{-2} e^{-\eta} \quad (2.229)$$

$$= \frac{e^{-\eta}}{1 + e^{-\eta}} \frac{1}{1 + e^{-\eta}} = \frac{1}{e^\eta + 1} \frac{1}{1 + e^{-\eta}} = (1 - \mu)\mu \quad (2.230)$$

2.4.3.3 Connection with the Fisher information matrix

In Section 3.3.4, we show that, under some regularity conditions, the **Fisher information matrix** is given by

$$\mathbf{F}(\eta) \triangleq \mathbb{E}_{p(\mathbf{x}|\eta)} [\nabla \log p(\mathbf{x}|\eta) \nabla \log p(\mathbf{x}|\eta)^T] = -\mathbb{E}_{p(\mathbf{x}|\eta)} [\nabla_{\boldsymbol{\eta}}^2 \log p(\mathbf{x}|\eta)] \quad (2.231)$$

Hence for an exponential family model we have

$$\mathbf{F}(\eta) = -\mathbb{E}_{p(\mathbf{x}|\eta)} [\nabla_{\boldsymbol{\eta}}^2 (\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta}))] = \nabla_{\boldsymbol{\eta}}^2 A(\eta) = \text{Cov} [\mathcal{T}(\mathbf{x})] \quad (2.232)$$

Thus the Hessian of the log partition function is the same as the FIM, which is the same as the covariance of the sufficient statistics. See Section 3.3.4.6 for details.

2.4.4 Canonical (natural) vs mean (moment) parameters

Let Ω be the set of normalizable natural parameters:

$$\Omega \triangleq \{\boldsymbol{\eta} \in \mathbb{R}^K : Z(\boldsymbol{\eta}) < \infty\} \quad (2.233)$$

We say that an exponential family is **regular** if Ω is an open set. It can be shown that Ω is a convex set, and $A(\boldsymbol{\eta})$ is a convex function defined over this set.

In Section 2.4.3, we prove that the derivative of the log partition function is equal to the mean of the sufficient statistics, i.e.,

$$\mathbf{m} = \mathbb{E} [\mathcal{T}(\mathbf{x})] = \nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) \quad (2.234)$$

The set of valid moment parameters is given by

$$\mathcal{M} = \{\mathbf{m} \in \mathbb{R}^K : \mathbb{E}_p [\mathcal{T}(\mathbf{x})] = \mathbf{m}\} \quad (2.235)$$

for some distribution p .

We have seen that we can convert from the natural parameters to the moment parameters using

$$\boldsymbol{\eta} = \nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) \quad (2.236)$$

If the family is minimal, one can show that

$$\boldsymbol{\eta} = \nabla_{\mathbf{m}} A^*(\mathbf{m}) \quad (2.237)$$

where $A^*(\mathbf{m})$ is the convex conjugate of A :

$$A^*(\mathbf{m}) \triangleq \sup_{\boldsymbol{\eta} \in \Omega} \boldsymbol{\mu}^T \boldsymbol{\eta} - A(\boldsymbol{\eta}) \quad (2.238)$$

Thus the pair of operators $(\nabla A, \nabla A^*)$ lets us go back and forth between the natural parameters $\boldsymbol{\eta} \in \Omega$ and the mean parameters $\mathbf{m} \in \mathcal{M}$.

For future reference, note that the Bregman divergences (Section 5.1.10) associated with A and A^* are as follows:

$$B_A(\boldsymbol{\lambda}_1 || \boldsymbol{\lambda}_2) = A(\boldsymbol{\lambda}_1) - A(\boldsymbol{\lambda}_2) - (\boldsymbol{\lambda}_1 - \boldsymbol{\lambda}_2)^T \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda}_2) \quad (2.239)$$

$$B_{A^*}(\boldsymbol{\mu}_1 || \boldsymbol{\mu}_2) = A(\boldsymbol{\mu}_1) - A(\boldsymbol{\mu}_2) - (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T \nabla_{\boldsymbol{\mu}} A(\boldsymbol{\mu}_2) \quad (2.240)$$

$$(2.241)$$

2.4.5 MLE for the exponential family

The likelihood of an exponential family model has the form

$$p(\mathcal{D}|\boldsymbol{\eta}) = \left[\prod_{n=1}^N h(\mathbf{x}_n) \right] \exp \left(\boldsymbol{\eta}^\top [\sum_{n=1}^N \mathcal{T}(\mathbf{x}_n)] - NA(\boldsymbol{\eta}) \right) \propto \exp [\boldsymbol{\eta}^\top \mathcal{T}(\mathcal{D}) - NA(\boldsymbol{\eta})] \quad (2.242)$$

where $\mathcal{T}(\mathcal{D})$ are the sufficient statistics:

$$\mathcal{T}(\mathcal{D}) = [\sum_{n=1}^N \mathcal{T}_1(\mathbf{x}_n), \dots, \sum_{n=1}^N \mathcal{T}_K(\mathbf{x}_n)] \quad (2.243)$$

For example, for the Bernoulli model we have $\mathcal{T}(\mathcal{D}) = [\sum_n \mathbb{I}(x_n = 1)]$, and for the univariate Gaussian, we have $\mathcal{T}(\mathcal{D}) = [\sum_n x_n, \sum_n x_n^2]$.

The **Pitman-Koopman-Darmois theorem** states that, under certain regularity conditions, the exponential family is the only family of distributions with finite sufficient statistics. (Here, finite means a size independent of the size of the dataset.) In other words, for an exponential family with natural parameters $\boldsymbol{\eta}$, we have

$$p(\mathcal{D}|\boldsymbol{\eta}) = p(\mathcal{T}(\mathcal{D})|\boldsymbol{\eta}) \quad (2.244)$$

We now show how to use this result to compute the MLE. The log likelihood is given by

$$\log p(\mathcal{D}|\boldsymbol{\eta}) = \boldsymbol{\eta}^\top \mathcal{T}(\mathcal{D}) - NA(\boldsymbol{\eta}) + \text{const} \quad (2.245)$$

Since $-A(\boldsymbol{\eta})$ is concave in $\boldsymbol{\eta}$, and $\boldsymbol{\eta}^\top \mathcal{T}(\mathcal{D})$ is linear in $\boldsymbol{\eta}$, we see that the log likelihood is concave, and hence has a unique global maximum. To derive this maximum, we use the fact (shown in Section 2.4.3) that the derivative of the log partition function yields the expected value of the sufficient statistic vector:

$$\nabla_{\boldsymbol{\eta}} \log p(\mathcal{D}|\boldsymbol{\eta}) = \nabla_{\boldsymbol{\eta}} \boldsymbol{\eta}^\top \mathcal{T}(\mathcal{D}) - N \nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) = \mathcal{T}(\mathcal{D}) - N \mathbb{E}[\mathcal{T}(\mathbf{x})] \quad (2.246)$$

For a single data case, this becomes

$$\nabla_{\boldsymbol{\eta}} \log p(\mathbf{x}|\boldsymbol{\eta}) = \mathcal{T}(\mathbf{x}) - \mathbb{E}[\mathcal{T}(\mathbf{x})] \quad (2.247)$$

Setting the gradient in Equation (2.246) to zero, we see that at the MLE, the empirical average of the sufficient statistics must equal the model's theoretical expected sufficient statistics, i.e., $\hat{\boldsymbol{\eta}}$ must satisfy

$$\mathbb{E}[\mathcal{T}(\mathbf{x})] = \frac{1}{N} \sum_{n=1}^N \mathcal{T}(\mathbf{x}_n) \quad (2.248)$$

This is called **moment matching**. For example, in the Bernoulli distribution, we have $\mathcal{T}(x) = \mathbb{I}(X = 1)$, so the MLE satisfies

$$\mathbb{E}[\mathcal{T}(x)] = p(X = 1) = \mu = \frac{1}{N} \sum_{n=1}^N \mathbb{I}(x_n = 1) \quad (2.249)$$

2.4.6 Exponential dispersion family

In this section, we consider a slight extension of the natural exponential family known as the **exponential dispersion family**. This will be useful when we discuss GLMs in Section 15.1. For a scalar variable, this has the form

$$p(x|\eta, \sigma^2) = h(x, \sigma^2) \exp \left[\frac{\eta x - A(\eta)}{\sigma^2} \right] \quad (2.250)$$

Here σ^2 is called the **dispersion parameter**. For fixed σ^2 , this is a natural exponential family.

2.4.7 Maximum entropy derivation of the exponential family

Suppose we want to find a distribution $p(\mathbf{x})$ to describe some data, where all we know are the expected values (F_k) of certain features or functions $f_k(\mathbf{x})$:

$$\int d\mathbf{x} p(\mathbf{x}) f_k(\mathbf{x}) = F_k \quad (2.251)$$

For example, f_1 might compute x , f_2 might compute x^2 , making F_1 the empirical mean and F_2 the empirical second moment. Our prior belief in the distribution is $q(\mathbf{x})$.

To formalize what we mean by “least number of assumptions”, we will search for the distribution that is as close as possible to our prior $q(\mathbf{x})$, in the sense of KL divergence (Section 5.1), while satisfying our constraints.

If we use a uniform prior, $q(\mathbf{x}) \propto 1$, minimizing the KL divergence is equivalent to maximizing the entropy (Section 5.2). The result is called a **maximum entropy model**.

To minimize KL subject to the constraints in Equation (2.251), and the constraint that $p(\mathbf{x}) \geq 0$ and $\sum_{\mathbf{x}} p(\mathbf{x}) = 1$, we need to use Lagrange multipliers. The Lagrangian is given by

$$J(p, \boldsymbol{\lambda}) = - \sum_{\mathbf{x}} p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} + \lambda_0 \left(1 - \sum_{\mathbf{x}} p(\mathbf{x}) \right) + \sum_k \lambda_k \left(F_k - \sum_{\mathbf{x}} p(\mathbf{x}) f_k(\mathbf{x}) \right) \quad (2.252)$$

We can use the calculus of variations to take derivatives wrt the function p , but we will adopt a simpler approach and treat \mathbf{p} as a fixed length vector (since we are assuming that \mathbf{x} is discrete). Then we have

$$\frac{\partial J}{\partial p_c} = -1 - \log \frac{p(x=c)}{q(x=c)} - \lambda_0 - \sum_k \lambda_k f_k(x=c) \quad (2.253)$$

Setting $\frac{\partial J}{\partial p_c} = 0$ for each c yields

$$p(\mathbf{x}) = \frac{q(\mathbf{x})}{Z} \exp \left(- \sum_k \lambda_k f_k(\mathbf{x}) \right) \quad (2.254)$$

where we have defined $Z \triangleq e^{1+\lambda_0}$. Using the sum-to-one constraint, we have

$$1 = \sum_{\mathbf{x}} p(\mathbf{x}) = \frac{1}{Z} \sum_{\mathbf{x}} q(\mathbf{x}) \exp \left(- \sum_k \lambda_k f_k(\mathbf{x}) \right) \quad (2.255)$$

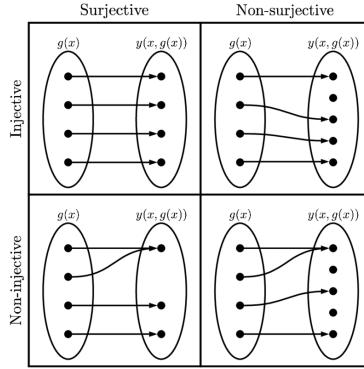


Figure 2.11: Illustration of injective and surjective functions.

Hence the normalization constant is given by

$$Z = \sum_{\mathbf{x}} q(\mathbf{x}) \exp \left(- \sum_k \lambda_k f_k(\mathbf{x}) \right) \quad (2.256)$$

This has exactly the form of the exponential family, where $\mathbf{f}(\mathbf{x})$ is the vector of sufficient statistics, $-\boldsymbol{\lambda}$ are the natural parameters, and $q(\mathbf{x})$ is our base measure.

For example, if the features are $f_1(x) = x$ and $f_2(x) = x^2$, and we want to match the first and second moments, we get the Gaussian distribution.

2.5 Transformations of random variables

Suppose $\mathbf{x} \sim p_x(\mathbf{x})$ is some random variable, and $\mathbf{y} = f(\mathbf{x})$ is some deterministic transformation of it. In this section, we discuss how to compute $p_y(\mathbf{y})$.

2.5.1 Invertible transformations (bijections)

Let f be a **bijection** that maps \mathbb{R}^n to \mathbb{R}^n . (A bijection is a function that is **injective**, or **one-to-one**, and **surjective**, as illustrated in Figure 2.11; this means that the function has a well-defined inverse.) Suppose we want to compute the pdf of $\mathbf{y} = f(\mathbf{x})$. The **change of variables** formula tells us that

$$p_y(\mathbf{y}) = p_x(f^{-1}(\mathbf{y})) |\det [\mathbf{J}_{f^{-1}}(\mathbf{y})]| \quad (2.257)$$

where $\mathbf{J}_{f^{-1}}(\mathbf{y})$ is the Jacobian of the inverse mapping f^{-1} evaluated at \mathbf{y} , and $|\det \mathbf{J}|$ is the absolute value of the determinant of \mathbf{J} . In other words,

$$\mathbf{J}_{f^{-1}}(\mathbf{y}) = \begin{pmatrix} \frac{\partial x_1}{\partial y_1} & \dots & \frac{\partial x_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial y_1} & \dots & \frac{\partial x_n}{\partial y_n} \end{pmatrix} \quad (2.258)$$

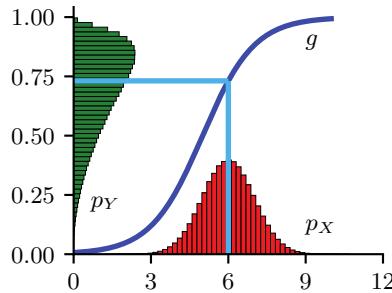


Figure 2.12: Example of the transformation of a density under a nonlinear transform. Note how the mode of the transformed distribution is not the transform of the original mode. Adapted from Exercise 1.4 of [Bis06]. Generated by [bayes_change_of_var.ipynb](#).

If the Jacobian matrix is triangular, the determinant reduces to a product of the terms on the main diagonal:

$$\det(\mathbf{J}) = \prod_{i=1}^n \frac{\partial x_i}{\partial y_i} \quad (2.259)$$

2.5.2 Monte Carlo approximation

Sometime it is difficult to compute the Jacobian. In this case, we can make a Monte Carlo approximation, by drawing S samples $\mathbf{x}^s \sim p(\mathbf{x})$, computing $\mathbf{y}^s = f(\mathbf{x}^s)$, and then constructing the empirical pdf

$$p_{\mathcal{D}}(\mathbf{y}) = \frac{1}{S} \sum_{s=1}^S \delta(\mathbf{y} - \mathbf{y}^s) \quad (2.260)$$

For example, let $x \sim \mathcal{N}(6, 1)$ and $y = f(x)$, where $f(x) = \frac{1}{1 + \exp(-x+5)}$. We can approximate $p(y)$ using Monte Carlo, as shown in Figure 2.12.

2.5.3 Probability integral transform

Suppose that X is a random variable with cdf P_X . Let $Y(X) = P_X(X)$ be a transformation of X . We now show that Y has a uniform distribution, a result known as the **probability integral transform** (PIT):

$$P_Y(y) = \Pr(Y \leq y) = \Pr(P_X(X) \leq y) \quad (2.261)$$

$$= \Pr(X \leq P_X^{-1}(y)) = P_X(P_X^{-1}(y)) = y \quad (2.262)$$

For example, in Figure 2.13, we show various distributions with pdf's p_X on the left column. We sample from these, to get $x_n \sim p_x$. Next we compute the empirical cdf of $Y = P_X(X)$, by computing $y_n = P_X(x_n)$ and then sorting the values; the results, shown in the middle column, show that this

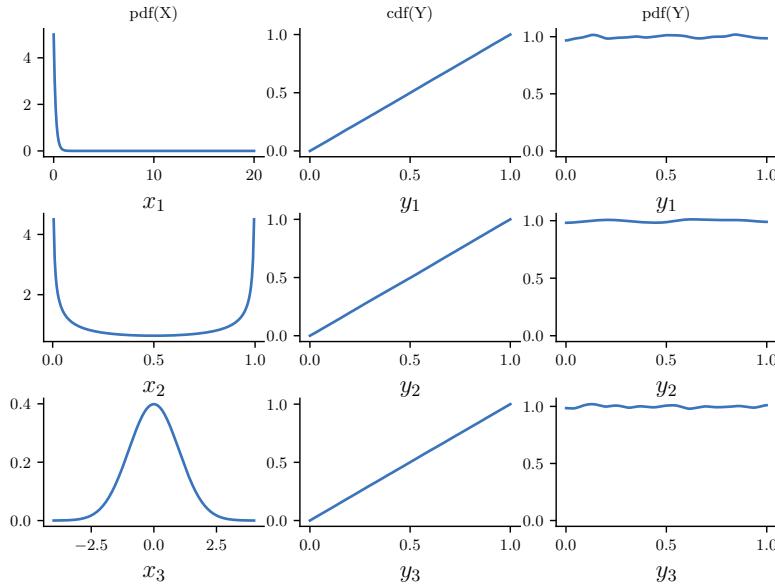


Figure 2.13: Illustration of the probability integral transform. Left column: 3 different pdf's for $p(X)$ from which we sample $x_n \sim p(x)$. Middle column: empirical cdf of $y_n = P_X(x_n)$. Right column: empirical pdf of $p(y_n)$ using a kernel density estimate. Adapted from Figure 11.17 of [MKL11]. Generated by [ecdf_sample.ipynb](#).

distribution is uniform. We can also approximate the pdf of Y by using kernel density estimation; this is shown in the right column, and we see that it is (approximately) flat.

We can use the PIT to test if a set of samples come from a given distribution using the **Kolmogorov–Smirnov test**. To do this, we plot the empirical cdf of the samples and the theoretical cdf of the distribution, and compute the maximum distance between these two curves, as illustrated in Figure 2.14. Formally, the KS statistic is defined as

$$D_n = \max_x |P_n(x) - P(x)| \quad (2.263)$$

where n is the sample size, P_n is the empirical cdf, and P is the theoretical cdf. The value D_n should approach 0 (as $n \rightarrow \infty$) if the samples are drawn from P .

Another application of the PIT is to generate samples from a distribution: if we have a way to sample from a uniform distribution, $u_n \sim \text{Unif}(0, 1)$, we can convert this to samples from any other distribution with cdf P_X by setting $x_n = P_X^{-1}(u_n)$.

2.6 Markov chains

Suppose that \mathbf{x}_t captures all the relevant information about the state of the system. This means it is a **sufficient statistic** for predicting the future given the past, i.e.,

$$p(\mathbf{x}_{t+\tau} | \mathbf{x}_t, \mathbf{x}_{1:t-1}) = p(\mathbf{x}_{t+\tau} | \mathbf{x}_t) \quad (2.264)$$

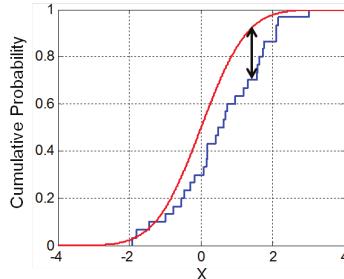


Figure 2.14: Illustration of the Kolmogorov-Smirnov statistic. The red line is a model cdf, the blue line is an empirical cdf, and the black arrow is the K-S statistic. From https://en.wikipedia.org/wiki/Kolmogorov_Smirnov_test. Used with kind permission of Wikipedia author Bscan.

for any $\tau \geq 0$. This is called the **Markov assumption**. In this case, we can write the joint distribution for any finite length sequence as follows:

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_2)p(\mathbf{x}_4|\mathbf{x}_3)\dots = p(\mathbf{x}_1) \prod_{t=2}^T p(\mathbf{x}_t|\mathbf{x}_{t-1}) \quad (2.265)$$

This is called a **Markov chain** or **Markov model**. Below we cover some of the basics of this topic; more details on the theory can be found in [Kun20].

2.6.1 Parameterization

In this section, we discuss how to represent a Markov model parametrically.

2.6.1.1 Markov transition kernels

The conditional distribution $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is called the **transition function**, **transition kernel**, or **Markov kernel**. This is just a conditional distribution over the states at time t given the state at time $t-1$, and hence it satisfies the conditions $p(\mathbf{x}_t|\mathbf{x}_{t-1}) \geq 0$ and $\int_{\mathbf{x} \in \mathcal{X}} d\mathbf{x} p(\mathbf{x}_t = \mathbf{x}|\mathbf{x}_{t-1}) = 1$.

If we assume the transition function $p(\mathbf{x}_t|\mathbf{x}_{1:t-1})$ is independent of time, then the model is said to be **homogeneous**, **stationary**, or **time-invariant**. This is an example of **parameter tying**, since the same parameter is shared by multiple variables. This assumption allows us to model an arbitrary number of variables using a fixed number of parameters. We will make the time-invariant assumption throughout the rest of this section.

2.6.1.2 Markov transition matrices

In this section, we assume that the variables are discrete, so $X_t \in \{1, \dots, K\}$. This is called a **finite-state Markov chain**. In this case, the conditional distribution $p(X_t|X_{t-1})$ can be written as a $K \times K$ matrix \mathbf{A} , known as the **transition matrix**, where $A_{ij} = p(X_t = j|X_{t-1} = i)$ is the probability of going from state i to state j . Each row of the matrix sums to one, $\sum_j A_{ij} = 1$, so this is called a **stochastic matrix**.

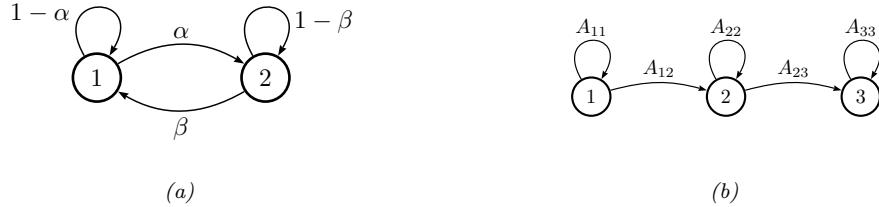


Figure 2.15: State transition diagrams for some simple Markov chains. Left: a 2-state chain. Right: a 3-state left-to-right chain.

A stationary, finite-state Markov chain is equivalent to a **stochastic automaton**. It is common to visualize such automata by drawing a directed graph, where nodes represent states and arrows represent legal transitions, i.e., non-zero elements of \mathbf{A} . This is known as a **state transition diagram**. The weights associated with the arcs are the probabilities. For example, the following 2-state chain

$$\mathbf{A} = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \quad (2.266)$$

is illustrated in Figure 2.15(a). The following 3-state chain

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & 0 \\ 0 & A_{22} & A_{23} \\ 0 & 0 & 1 \end{pmatrix} \quad (2.267)$$

is illustrated in Figure 2.15(b). This is called a **left-to-right transition matrix**.

The A_{ij} element of the transition matrix specifies the probability of getting from i to j in one step. The n -step transition matrix $\mathbf{A}(n)$ is defined as

$$A_{ij}(n) \triangleq p(X_{t+n} = j | X_t = i) \quad (2.268)$$

which is the probability of getting from i to j in exactly n steps. Obviously $\mathbf{A}(1) = \mathbf{A}$. The **Chapman-Kolmogorov** equations state that

$$A_{ij}(m+n) = \sum_{k=1}^K A_{ik}(m) A_{kj}(n) \quad (2.269)$$

In words, the probability of getting from i to j in $m + n$ steps is just the probability of getting from i to k in m steps, and then from k to j in n steps, summed up over all k . We can write the above as a matrix multiplication

$$\mathbf{A}(m+n) = \mathbf{A}(m)\mathbf{A}(n) \quad (2.270)$$

Hence

$$\mathbf{A}(n) = \mathbf{A} \quad \mathbf{A}(n-1) = \mathbf{A} \quad \mathbf{A}(n-2) = \cdots = \mathbf{A}^n \quad (2.271)$$

Thus we can simulate multiple steps of a Markov chain by “powering up” the transition matrix.

christians first inhabit wherein thou hast forgive if a man childless and of laying of core these
are the heavens shall reel to and fro to seek god they set their horses and children of israel

Figure 2.16: Example output from an 10-gram character-level Markov model trained on the King James Bible. The prefix “christians” is given to the model. Generated by [ngram_character_demo.ipynb](#).

2.6.1.3 Higher-order Markov models

The first-order Markov assumption is rather strong. Fortunately, we can easily generalize first-order models to depend on the last n observations, thus creating a model of order (memory length) n :

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_{1:n}) \prod_{t=n+1}^T p(\mathbf{x}_t | \mathbf{x}_{t-n:t-1}) \quad (2.272)$$

This is called a **Markov model of order n** . If $n = 1$, this is called a **bigram model**, since we need to represent pairs of characters, $p(\mathbf{x}_t | \mathbf{x}_{t-1})$. If $n = 2$, this is called a **trigram model**, since we need to represent triples of characters, $p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2})$. In general, this is called an **n-gram model**.

Note, however, we can always convert a higher order Markov model to a first order one by defining an augmented state space that contains the past n observations. For example, if $n = 2$, we define $\tilde{\mathbf{x}}_t = (\mathbf{x}_{t-1}, \mathbf{x}_t)$ and use

$$p(\tilde{\mathbf{x}}_{1:T}) = p(\tilde{\mathbf{x}}_2) \prod_{t=3}^T p(\tilde{\mathbf{x}}_t | \tilde{\mathbf{x}}_{t-1}) = p(\mathbf{x}_1, \mathbf{x}_2) \prod_{t=3}^T p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}) \quad (2.273)$$

Therefore we will just focus on first-order models throughout the rest of this section.

2.6.2 Application: language modeling

One important application of Markov models is to create **language models (LM)**, which are models which can generate (or score) a sequence of words. When we use a finite-state Markov model with a memory of length $m = n - 1$, it is called an **n-gram model**. For example, if $m = 1$, we get a **unigram model** (no dependence on previous words); if $m = 2$, we get a **bigram model** (depends on previous word); if $m = 3$, we get a **trigram model** (depends on previous two words); etc. See Figure 2.16 for some generated text.

These days, most LMs are built using recurrent neural nets (see Section 16.3.4), which have unbounded memory. However, simple n-gram models can still do quite well when trained with enough data [Che17].

Language models have various applications, such as priors for spelling correction (see Section 29.3.3) or automatic speech recognition. In addition, conditional language models can be used to generate sequences given inputs, such as mapping one language to another, or an image to a sequence, etc.

2.6.3 Parameter estimation

In this section, we discuss how to estimate the parameters of a Markov model.

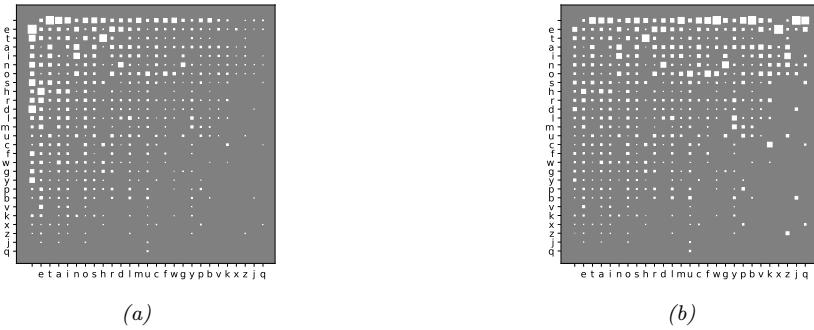


Figure 2.17: (a) **Hinton diagram** showing character bigram counts as estimated from H. G. Wells’s book The Time Machine. Characters are sorted in decreasing unigram frequency; the first one is a space character. The most frequent bigram is ‘e-’, where - represents space. (b) Same as (a) but each row is normalized across the columns. Generated by `bigram_hinton_diagram.ipynb`.

2.6.3.1 Maximum likelihood estimation

The probability of any particular sequence of length T is given by

$$p(x_{1:T}|\boldsymbol{\theta}) = \pi(x_1)A(x_1, x_2) \dots A(x_{T-1}, x_T) \quad (2.274)$$

$$= \prod_{j=1}^K (\pi_j)^{\mathbb{I}(x_1=j)} \prod_{t=2}^T \prod_{j=1}^K \prod_{k=1}^K (A_{jk})^{\mathbb{I}(x_t=k, x_{t-1}=j)} \quad (2.275)$$

Hence the log-likelihood of a set of sequences $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, where $\mathbf{x}_i = (x_{i1}, \dots, x_{iT_i})$ is a sequence of length T_i , is given by

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^N \log p(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_j N_j^1 \log \pi_j + \sum_j \sum_k N_{jk} \log A_{jk} \quad (2.276)$$

where we define the following counts:

$$N_j^1 \triangleq \sum_{i=1}^N \mathbb{I}(x_{i1} = j), \quad N_{jk} \triangleq \sum_{i=1}^N \sum_{t=1}^{T_i-1} \mathbb{I}(x_{i,t} = j, x_{i,t+1} = k), \quad N_j = \sum_k N_{jk} \quad (2.277)$$

By adding Lagrange multipliers to enforce the sum to one constraints, one can show (see e.g., [Mur22, Sec 4.2.4]) that the MLE is given by the normalized counts:

$$\hat{\pi}_j = \frac{N_j^1}{\sum_{j'} N_{j'}^1}, \quad \hat{A}_{jk} = \frac{N_{jk}}{N_j} \quad (2.278)$$

We often replace N_j^1 , which is how often symbol j is seen at the start of a sequence, by N_j , which is how often symbol j is seen anywhere in a sequence. This lets us estimate parameters from a single sequence.

The counts N_j are known as **unigram statistics**, and N_{jk} are known as **bigram statistics**. For example, Figure 2.17 shows some 2-gram counts for the characters $\{a, \dots, z, -\}$ (where $-$ represents space) as estimated from H. G. Wells's book *The Time Machine*.

2.6.3.2 Sparse data problem

When we try to fit n-gram models for large n , we quickly encounter problems with overfitting due to data sparsity. To see that, note that many of the estimated counts N_{jk} will be 0, since now j indexes over discrete contexts of size K^{n-1} , which will become increasingly rare. Even for bigram models ($n = 2$), problems can arise if K is large. For example, if we have $K \sim 50,000$ words in our vocabulary, then a bi-gram model will have about 2.5 billion free parameters, corresponding to all possible word pairs. It is very unlikely we will see all of these in our training data. However, we do not want to predict that a particular word string is totally impossible just because we happen not to have seen it in our training text — that would be a severe form of overfitting.⁹

A “brute force” solution to this problem is to gather lots and lots of data. For example, Google has fit n-gram models (for $n = 1 : 5$) based on one trillion words extracted from the web. Their data, which is over 100GB when uncompressed, is publically available.¹⁰ Although such an approach can be surprisingly successful (as discussed in [HNP09]), it is rather unsatisfying, since humans are able to learn language from much less data (see e.g., [TX00]).

2.6.3.3 MAP estimation

A simple solution to the sparse data problem is to use MAP estimation with a uniform Dirichlet prior, $\mathbf{A}_{j:} \sim \text{Dir}(\alpha \mathbf{1})$. In this case, the MAP estimate becomes

$$\hat{A}_{jk} = \frac{N_{jk} + \alpha}{N_j + K\alpha} \quad (2.279)$$

If $\alpha = 1$, this is called **add-one smoothing**.

The main problem with add-one smoothing is that it assumes that all n-grams are equally likely, which is not very realistic. We discuss a more sophisticated approach, based on hierarchical Bayes, in Section 3.7.3.

2.6.4 Stationary distribution of a Markov chain

Suppose we continually draw consecutive samples from a Markov chain. In the case of a finite state space, we can think of this as “hopping” from one state to another. We will tend to spend more time in some states than others, depending on the transition graph. The long term distribution over states is known as the **stationary distribution** of the chain. In this section, we discuss some of the relevant theory. In Chapter 12, we discuss an important application, known as MCMC, which is a way to generate samples from hard-to-normalize probability distributions. In Supplementary Section 2.2

9. A famous example of an improbable, but syntactically valid, English word string, due to Noam Chomsky [Cho57], is “colourless green ideas sleep furiously”. We would not want our model to predict that this string is impossible. Even ungrammatical constructs should be allowed by our model with a certain probability, since people frequently violate grammatical rules, especially in spoken language.

10. See <http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html> for details.

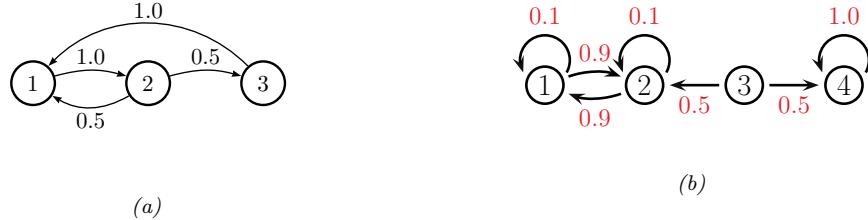


Figure 2.18: Some Markov chains. (a) A 3-state aperiodic chain. (b) A reducible 4-state chain.

we consider Google's PageRank algorithm for ranking web pages, which also leverages the concept of stationary distributions.

2.6.4.1 What is a stationary distribution?

Let $A_{ij} = p(X_t = j | X_{t-1} = i)$ be the one-step transition matrix, and let $\pi_t(j) = p(X_t = j)$ be the probability of being in state j at time t .

If we have an initial distribution over states of $\boldsymbol{\pi}_0$, then at time 1 we have

$$\pi_1(j) = \sum_i \pi_0(i) A_{ij} \quad (2.280)$$

or, in matrix notation, $\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{A}$, where we have followed the standard convention of assuming $\boldsymbol{\pi}$ is a *row* vector, so we post-multiply by the transition matrix.

Now imagine iterating these equations. If we ever reach a stage where $\boldsymbol{\pi} = \boldsymbol{\pi} \mathbf{A}$, then we say we have reached the **stationary distribution** (also called the **invariant distribution** or **equilibrium distribution**). Once we enter the stationary distribution, we will never leave.

For example, consider the chain in Figure 2.18(a). To find its stationary distribution, we write

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (\pi_1 \quad \pi_2 \quad \pi_3) \begin{pmatrix} 1 - A_{12} - A_{13} & A_{12} & A_{13} \\ A_{21} & 1 - A_{21} - A_{23} & A_{23} \\ A_{31} & A_{32} & 1 - A_{31} - A_{32} \end{pmatrix} \quad (2.281)$$

Hence $\pi_1(A_{12} + A_{13}) = \pi_2 A_{21} + \pi_3 A_{31}$. In general, we have

$$\pi_i \sum_{j \neq i} A_{ij} = \sum_{j \neq i} \pi_j A_{ji} \quad (2.282)$$

In other words, the probability of being in state i times the net flow out of state i must equal the probability of being in each other state j times the net flow from that state into i . These are called the **global balance equations**. We can then solve these equations, subject to the constraint that $\sum_j \pi_j = 1$, to find the stationary distribution, as we discuss below.

2.6.4.2 Computing the stationary distribution

To find the stationary distribution, we can just solve the eigenvector equation $\mathbf{A}^\top \mathbf{v} = \mathbf{v}$, and then to set $\boldsymbol{\pi} = \mathbf{v}^\top$, where \mathbf{v} is an eigenvector with eigenvalue 1. (We can be sure such an eigenvector

exists, since \mathbf{A} is a row-stochastic matrix, so $\mathbf{A}\mathbf{1} = \mathbf{1}$; also recall that the eigenvalues of \mathbf{A} and \mathbf{A}^T are the same.) Of course, since eigenvectors are unique only up to constants of proportionality, we must normalize \mathbf{v} at the end to ensure it sums to one.

Note, however, that the eigenvectors are only guaranteed to be real-valued if all entries in the matrix are strictly positive, $A_{ij} > 0$ (and hence $A_{ij} < 1$, due to the sum-to-one constraint). A more general approach, which can handle chains where some transition probabilities are 0 or 1 (such as Figure 2.18(a)), is as follows. We have K constraints from $\boldsymbol{\pi}(\mathbf{I} - \mathbf{A}) = \mathbf{0}_{K \times 1}$ and 1 constraint from $\boldsymbol{\pi}\mathbf{1}_{K \times 1} = 1$. Hence we have to solve $\boldsymbol{\pi}\mathbf{M} = \mathbf{r}$, where $\mathbf{M} = [\mathbf{I} - \mathbf{A}, \mathbf{1}]$ is a $K \times (K + 1)$ matrix, and $\mathbf{r} = [0, 0, \dots, 0, 1]$ is a $1 \times (K + 1)$ vector. However, this is overconstrained, so we will drop the last column of $\mathbf{I} - \mathbf{A}$ in our definition of \mathbf{M} , and drop the last 0 from \mathbf{r} . For example, for a 3 state chain we have to solve this linear system:

$$(\pi_1 \quad \pi_2 \quad \pi_3) \begin{pmatrix} 1 - A_{11} & -A_{12} & 1 \\ -A_{21} & 1 - A_{22} & 1 \\ -A_{31} & -A_{32} & 1 \end{pmatrix} = (0 \quad 0 \quad 1) \quad (2.283)$$

For the chain in Figure 2.18(a) we find $\boldsymbol{\pi} = [0.4, 0.4, 0.2]$. We can easily verify this is correct, since $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A}$.

Unfortunately, not all chains have a stationary distribution, as we explain below.

2.6.4.3 When does a stationary distribution exist?

Consider the 4-state chain in Figure 2.18(b). If we start in state 4, we will stay there forever, since 4 is an **absorbing state**. Thus $\boldsymbol{\pi} = (0, 0, 0, 1)$ is one possible stationary distribution. However, if we start in 1 or 2, we will oscillate between those two states forever. So $\boldsymbol{\pi} = (0.5, 0.5, 0, 0)$ is another possible stationary distribution. If we start in state 3, we could end up in either of the above stationary distributions with equal probability. The corresponding transition graph has two disjoint connected components.

We see from this example that a necessary condition to have a unique stationary distribution is that the state transition diagram be a singly connected component, i.e., we can get from any state to any other state. Such chains are called **irreducible**.

Now consider the 2-state chain in Figure 2.15(a). This is irreducible provided $\alpha, \beta > 0$. Suppose $\alpha = \beta = 0.9$. It is clear by symmetry that this chain will spend 50% of its time in each state. Thus $\boldsymbol{\pi} = (0.5, 0.5)$. But now suppose $\alpha = \beta = 1$. In this case, the chain will oscillate between the two states, but the long-term distribution on states depends on where you start from. If we start in state 1, then on every odd time step (1, 3, 5, ...) we will be in state 1; but if we start in state 2, then on every odd time step we will be in state 2.

This example motivates the following definition. Let us say that a chain has a **limiting distribution** if $\pi_j = \lim_{n \rightarrow \infty} A_{ij}^n$ exists and is independent of the starting state i , for all j . If this holds, then the long-run distribution over states will be independent of the starting state:

$$p(X_t = j) = \sum_i p(X_0 = i) A_{ij}(t) \rightarrow \pi_j \text{ as } t \rightarrow \infty \quad (2.284)$$

Let us now characterize when a limiting distribution exists. Define the **period** of state i to be $d(i) \triangleq \gcd\{t : A_{ii}(t) > 0\}$, where gcd stands for **greatest common divisor**, i.e., the largest integer

that divides all the members of the set. For example, in Figure 2.18(a), we have $d(1) = d(2) = \gcd(2, 3, 4, 6, \dots) = 1$ and $d(3) = \gcd(3, 5, 6, \dots) = 1$. We say a state i is **aperiodic** if $d(i) = 1$. (A sufficient condition to ensure this is if state i has a self-loop, but this is not a necessary condition.) We say a chain is aperiodic if all its states are aperiodic. One can show the following important result:

Theorem 2.6.1. *Every irreducible (singly connected), aperiodic finite state Markov chain has a limiting distribution, which is equal to π , its unique stationary distribution.*

A special case of this result says that every regular finite state chain has a unique stationary distribution, where a **regular** chain is one whose transition matrix satisfies $A_{ij}^n > 0$ for some integer n and all i, j , i.e., it is possible to get from any state to any other state in n steps. Consequently, after n steps, the chain could be in any state, no matter where it started. One can show that sufficient conditions to ensure regularity are that the chain be irreducible (singly connected) and that every state have a self-transition.

To handle the case of Markov chains whose state space is not finite (e.g, the countable set of all integers, or all the uncountable set of all reals), we need to generalize some of the earlier definitions. Since the details are rather technical, we just briefly state the main results without proof. See e.g., [GS92] for details.

For a stationary distribution to exist, we require irreducibility (singly connected) and aperiodicity, as before. But we also require that each state is **recurrent**, which means that you will return to that state with probability 1. As a simple example of a non-recurrent state (i.e., a **transient** state), consider Figure 2.18(b): state 3 is transient because one immediately leaves it and either spins around state 4 forever, or oscillates between states 1 and 2 forever. There is no way to return to state 3.

It is clear that any finite-state irreducible chain is recurrent, since you can always get back to where you started from. But now consider an example with an infinite state space. Suppose we perform a random walk on the integers, $\mathcal{X} = \{\dots, -2, -1, 0, 1, 2, \dots\}$. Let $A_{i,i+1} = p$ be the probability of moving right, and $A_{i,i-1} = 1 - p$ be the probability of moving left. Suppose we start at $X_1 = 0$. If $p > 0.5$, we will shoot off to $+\infty$; we are not guaranteed to return. Similarly, if $p < 0.5$, we will shoot off to $-\infty$. So in both cases, the chain is not recurrent, even though it is irreducible. If $p = 0.5$, we can return to the initial state with probability 1, so the chain is recurrent. However, the distribution keeps spreading out over a larger and larger set of the integers, so the expected time to return is infinite. This prevents the chain from having a stationary distribution.

More formally, we define a state to be **non-null recurrent** if the expected time to return to this state is finite. We say that a state is **ergodic** if it is aperiodic, recurrent, and non-null. We say that a chain is ergodic if all its states are ergodic. With these definitions, we can now state our main theorem:

Theorem 2.6.2. *Every irreducible, ergodic Markov chain has a limiting distribution, which is equal to π , its unique stationary distribution.*

This generalizes Theorem 2.6.1, since for irreducible finite-state chains, all states are recurrent and non-null.

2.6.4.4 Detailed balance

Establishing ergodicity can be difficult. We now give an alternative condition that is easier to verify.

We say that a Markov chain \mathbf{A} is **time reversible** if there exists a distribution $\boldsymbol{\pi}$ such that

$$\pi_i A_{ij} = \pi_j A_{ji} \quad (2.285)$$

These are called the **detailed balance equations**. This says that the flow from i to j must equal the flow from j to i , weighted by the appropriate source probabilities.

We have the following important result.

Theorem 2.6.3. *If a Markov chain with transition matrix \mathbf{A} is regular and satisfies the detailed balance equations wrt distribution $\boldsymbol{\pi}$, then $\boldsymbol{\pi}$ is a stationary distribution of the chain.*

Proof. To see this, note that

$$\sum_i \pi_i A_{ij} = \sum_i \pi_j A_{ji} = \pi_j \sum_i A_{ji} = \pi_j \quad (2.286)$$

and hence $\boldsymbol{\pi} = \mathbf{A}\boldsymbol{\pi}$. \square

Note that this condition is sufficient but not necessary (see Figure 2.18(a) for an example of a chain with a stationary distribution which does not satisfy detailed balance).

2.7 Divergence measures between probability distributions

In this section, we discuss various ways to compare two probability distributions, P and Q , defined on the same space. For example, suppose the distributions are defined in terms of samples, $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\} \sim P$ and $\mathcal{X}' = \{\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_M\} \sim Q$. Determining if the samples come from the same distribution is known as a **two-sample test** (see Figure 2.19 for an illustration). This can be computed by defining some suitable **divergence metric** $D(P, Q)$ and comparing it to a threshold. (We use the term “divergence” rather than distance since we will not require D to be symmetric.) Alternatively, suppose P is an empirical distribution of data, and Q is the distribution induced by a model. We can check how well the model approximates the data by comparing $D(P, Q)$ to a threshold; this is called a **goodness-of-fit** test.

There are two main ways to compute the divergence between a pair of distributions: in terms of their difference, $P - Q$ (see e.g., [Sug+13]) or in terms of their ratio, P/Q (see e.g., [SSK12]). We briefly discuss both of these below. (Our presentation is based, in part, on [GSJ19].)

2.7.1 f -divergence

In this section, we compare distributions in terms of their density ratio $r(\mathbf{x}) = p(\mathbf{x})/q(\mathbf{x})$. In particular, consider the **f -divergence** [Mor63; AS66; Csi67; LV06; CS04], which is defined as follows:

$$D_f(p||q) = \int q(\mathbf{x}) f\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) d\mathbf{x} \quad (2.287)$$

where $f : \mathbb{R}_+ \rightarrow \mathbb{R}$ is a convex function satisfying $f(1) = 0$. From Jensen’s inequality (Section 5.1.2.2), it follows that $D_f(p||q) \geq 0$, and obviously $D_f(p||p) = 0$, so D_f is a valid divergence. Below we discuss some important special cases of f -divergences. (Note that f -divergences are also called ϕ -divergences.)

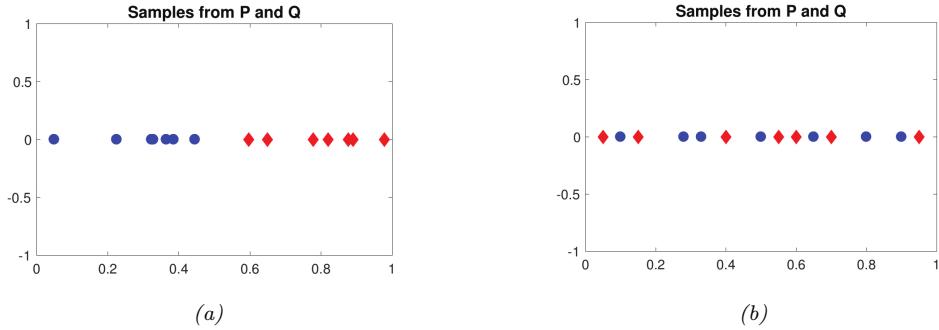


Figure 2.19: Samples from two distributions which are (a) different and (b) similar. From a figure from [GSJ19]. Used with kind permission of Arthur Gretton.

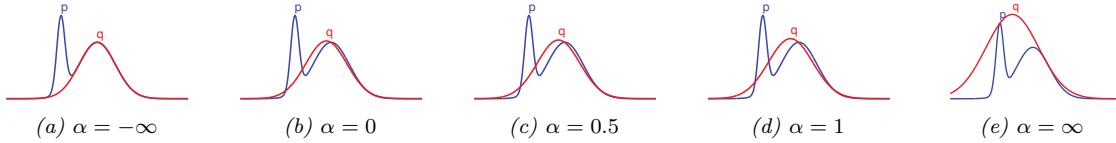


Figure 2.20: The Gaussian q which minimizes α -divergence to p (a mixture of two Gaussians), for varying α . From Figure 1 of [Min05]. Used with kind permission of Tom Minka.

2.7.1.1 KL divergence

Suppose we compute the f -divergence using $f(r) = r \log(r)$. In this case, we get a quantity called the **Kullback Leibler divergence**, defined as follows:

$$D_{\text{KL}}(p \parallel q) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \quad (2.288)$$

See Section 5.1 for more details.

2.7.1.2 Alpha divergence

If $f(x) = \frac{4}{1-\alpha^2}(1-x^{\frac{1+\alpha}{2}})$, the f -divergence becomes the **alpha divergence** [Ama09], which is as follows:

$$D_{\alpha}^A(p \parallel q) \triangleq \frac{4}{1-\alpha^2} \left(1 - \int p(\mathbf{x})^{(1+\alpha)/2} q(\mathbf{x})^{(1-\alpha)/2} d\mathbf{x} \right) \quad (2.289)$$

where we assume $\alpha \neq \pm 1$. Another common parameterization, and the one used by Minka in [Min05], is as follows:

$$D_{\alpha}^M(p \parallel q) = \frac{1}{\alpha(1-\alpha)} \left(1 - \int p(\mathbf{x})^{\alpha} q(\mathbf{x})^{1-\alpha} d\mathbf{x} \right) \quad (2.290)$$

This can be converted to Amari's notation using $D_{\alpha'}^A = D_{\alpha}^M$ where $\alpha' = 2\alpha - 1$. (We will use the Minka convention.)

We see from Figure 2.20 that as $\alpha \rightarrow -\infty$, q prefers to match one mode of p , whereas when $\alpha \rightarrow \infty$, q prefers to cover all of p . More precisely, one can show that as $\alpha \rightarrow 0$, the alpha divergence tends towards $D_{\text{KL}}(q \parallel p)$, and as $\alpha \rightarrow 1$, the alpha divergence tends towards $D_{\text{KL}}(p \parallel q)$. Also, when $\alpha = 0.5$, the alpha divergence equals the Hellinger distance (Section 2.7.1.3).

2.7.1.3 Hellinger distance

The (squared) **Hellinger distance** is defined as follows:

$$D_H^2(p \parallel q) \triangleq \frac{1}{2} \int \left(p(\mathbf{x})^{\frac{1}{2}} - q(\mathbf{x})^{\frac{1}{2}} \right)^2 d\mathbf{x} = 1 - \int \sqrt{p(\mathbf{x})q(\mathbf{x})} d\mathbf{x} \quad (2.291)$$

This is a valid distance metric, since it is symmetric, nonnegative, and satisfies the triangle inequality.

We see that this is equal (up to constant factors) to the f -divergence with $f(r) = (\sqrt{r} - 1)^2$, since

$$\int d\mathbf{x} q(\mathbf{x}) \left(\frac{p^{\frac{1}{2}}(\mathbf{x})}{q^{\frac{1}{2}}(\mathbf{x})} - 1 \right)^2 = \int d\mathbf{x} q(\mathbf{x}) \left(\frac{p^{\frac{1}{2}}(\mathbf{x}) - q^{\frac{1}{2}}(\mathbf{x})}{q^{\frac{1}{2}}(\mathbf{x})} \right)^2 = \int d\mathbf{x} \left(p^{\frac{1}{2}}(\mathbf{x}) - q^{\frac{1}{2}}(\mathbf{x}) \right)^2 \quad (2.292)$$

2.7.1.4 Chi-squared distance

The **chi-squared distance** χ^2 is defined by

$$\chi^2(p, q) \triangleq \frac{1}{2} \int \frac{(q(\mathbf{x}) - p(\mathbf{x}))^2}{q(\mathbf{x})} d\mathbf{x} \quad (2.293)$$

This is equal (up to constant factors) to an f -divergence where $f(r) = (r - 1)^2$, since

$$\int d\mathbf{x} q(\mathbf{x}) \left(\frac{p(\mathbf{x})}{q(\mathbf{x})} - 1 \right)^2 = \int d\mathbf{x} q(\mathbf{x}) \left(\frac{p(\mathbf{x}) - q(\mathbf{x})}{q(\mathbf{x})} \right)^2 = \int d\mathbf{x} \frac{1}{q(\mathbf{x})} (p(\mathbf{x}) - q(\mathbf{x}))^2 \quad (2.294)$$

2.7.2 Integral probability metrics

In this section, we compute the divergence between two distributions in terms of $P - Q$ using an **integral probability metric** or **IPM** [Sri+09]. This is defined as follows:

$$D_{\mathcal{F}}(P, Q) \triangleq \sup_{f \in \mathcal{F}} |\mathbb{E}_{p(\mathbf{x})} [f(\mathbf{x})] - \mathbb{E}_{q(\mathbf{x}')} [f(\mathbf{x}')]| \quad (2.295)$$

where \mathcal{F} is some class of “smooth” functions. The function f that maximizes the difference between these two expectations is called the **witness function**. See Figure 2.21 for an illustration.

There are several ways to define the function class \mathcal{F} . One approach is to use an RKHS, defined in terms of a positive definite kernel function; this gives rise to the method known as maximum mean discrepancy or MMD. See Section 2.7.3 for details.

Another approach is to define \mathcal{F} to be the set of functions that have bounded Lipschitz constant, i.e., $\mathcal{F} = \{||f||_L \leq 1\}$, where

$$||f||_L = \sup_{\mathbf{x} \neq \mathbf{x}'} \frac{|f(\mathbf{x}) - f(\mathbf{x}')|}{\|\mathbf{x} - \mathbf{x}'\|} \quad (2.296)$$

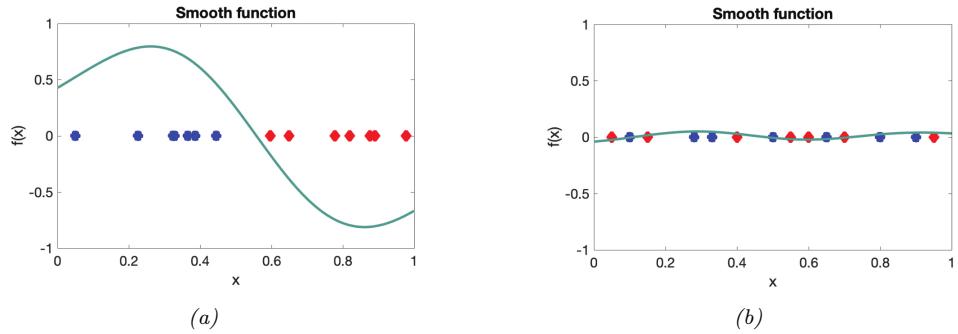


Figure 2.21: A smooth witness function for comparing two distributions which are (a) different and (b) similar. From a figure from [GSJ19]. Used with kind permission of Arthur Gretton.

The IPM in this case is equal to the **Wasserstein-1 distance**

$$W_1(P, Q) \triangleq \sup_{\|f\|_L \leq 1} |\mathbb{E}_{p(\mathbf{x})}[f(\mathbf{x})] - \mathbb{E}_{q(\mathbf{x}')}[f(\mathbf{x}')]| \quad (2.297)$$

See Section 6.8.2.4 for details.

2.7.3 Maximum mean discrepancy (MMD)

In this section, we describe the **maximum mean discrepancy** or **MMD** method of [Gre+12], which defines a discrepancy measure $D(P, Q)$ using samples from the two distributions. The samples are compared using positive definite kernels (Section 18.2), which can handle high-dimensional inputs. This approach can be used to define two-sample tests, and to train implicit generative models (Section 26.2.4).

2.7.3.1 MMD as an IPM

The MMD is an integral probability metric (Section 2.7.2) of the form

$$\text{MMD}(P, Q; \mathcal{F}) = \sup_{f \in \mathcal{F}: \|f\|_1 \leq 1} [\mathbb{E}_{p(\mathbf{x})} [f(\mathbf{x})] - \mathbb{E}_{q(\mathbf{x}')} [f(\mathbf{x}')]] \quad (2.298)$$

where \mathcal{F} is an RKHS (Section 18.3.7.1) defined by a positive definite kernel function \mathcal{K} . We can represent functions in this set as an infinite sum of basis functions

$$f(\mathbf{x}) = \langle f, \phi(\mathbf{x}) \rangle_{\mathcal{F}} = \sum_{l=1}^{\infty} f_l \phi_l(\mathbf{x}) \quad (2.299)$$

We restrict the set of witness functions f to be those that are in the unit ball of this RKHS, so $\|f\|_{\mathcal{F}}^2 = \sum_{l=1}^{\infty} f_l^2 \leq 1$. By the linearity of expectation, we have

$$\mathbb{E}_{p(\mathbf{x})} [f(\mathbf{x})] = \langle f, \mathbb{E}_{p(\mathbf{x})} [\phi(\mathbf{x})] \rangle_{\mathcal{F}} = \langle f, \boldsymbol{\mu}_P \rangle_{\mathcal{F}} \quad (2.300)$$

where $\boldsymbol{\mu}_P$ is called the **kernel mean embedding** of distribution P [Mua+17]. Hence

$$\text{MMD}(P, Q; \mathcal{F}) = \sup_{\|f\| \leq 1} \langle f, \boldsymbol{\mu}_P - \boldsymbol{\mu}_Q \rangle_{\mathcal{F}} = \frac{\boldsymbol{\mu}_P - \boldsymbol{\mu}_Q}{\|\boldsymbol{\mu}_P - \boldsymbol{\mu}_Q\|} \quad (2.301)$$

since the unit vector f that maximizes the inner product is parallel to the difference in feature means.

To get some intuition, suppose $\phi(x) = [x, x^2]$. In this case, the MMD computes the difference in the first two moments of the two distributions. This may not be enough to distinguish all possible distributions. However, using a Gaussian kernel is equivalent to comparing two infinitely large feature vectors, as we show in Section 18.2.6, and hence we are effectively comparing all the moments of the two distributions. Indeed, one can show that $\text{MMD}=0$ iff $P = Q$, provided we use a non-degenerate kernel.

2.7.3.2 Computing the MMD using the kernel trick

In this section, we describe how to compute Equation (2.301) in practice, given two sets of samples, $\mathcal{X} = \{\mathbf{x}_n\}_{n=1}^N$ and $\mathcal{X}' = \{\mathbf{x}'_m\}_{m=1}^M$, where $\mathbf{x}_n \sim P$ and $\mathbf{x}'_m \sim Q$. Let $\boldsymbol{\mu}_P = \frac{1}{N} \sum_{n=1}^N \phi(\mathbf{x}_n)$ and $\boldsymbol{\mu}_Q = \frac{1}{M} \sum_{m=1}^M \phi(\mathbf{x}'_m)$ be empirical estimates of the kernel mean embeddings of the two distributions. Then the squared MMD is given by

$$\text{MMD}^2(\mathcal{X}, \mathcal{X}') \triangleq \left\| \frac{1}{N} \sum_{n=1}^N \phi(\mathbf{x}_n) - \frac{1}{M} \sum_{m=1}^M \phi(\mathbf{x}'_m) \right\|^2 \quad (2.302)$$

$$\begin{aligned} &= \frac{1}{N^2} \sum_{n=1}^N \sum_{n'=1}^N \phi(\mathbf{x}_n)^T \phi(\mathbf{x}_{n'}) - \frac{2}{NM} \sum_{n=1}^N \sum_{m=1}^M \phi(\mathbf{x}_n)^T \phi(\mathbf{x}'_m) \\ &\quad + \frac{1}{M^2} \sum_{m=1}^M \sum_{m'=1}^M \phi(\mathbf{x}'_{m'})^T \phi(\mathbf{x}'_{m'}) \end{aligned} \quad (2.303)$$

Since Equation (2.303) only involves inner products of the feature vectors, we can use the kernel trick (Section 18.2.5) to rewrite the above as follows:

$$\text{MMD}^2(\mathcal{X}, \mathcal{X}') = \frac{1}{N^2} \sum_{n=1}^N \sum_{n'=1}^N \mathcal{K}(\mathbf{x}_n, \mathbf{x}_{n'}) - \frac{2}{NM} \sum_{n=1}^N \sum_{m=1}^M \mathcal{K}(\mathbf{x}_n, \mathbf{x}'_m) + \frac{1}{M^2} \sum_{m=1}^M \sum_{m'=1}^M \mathcal{K}(\mathbf{x}'_m, \mathbf{x}'_{m'}) \quad (2.304)$$

2.7.3.3 Linear time computation

The MMD takes $O(N^2)$ time to compute, where N is the number of samples from each distribution. In [Chw+15], they present a different test statistic called the **unnormalized mean embedding** or **UME**, that can be computed in $O(N)$ time.

The key idea is to notice that evaluating

$$\text{witness}^2(\mathbf{v}) = (\boldsymbol{\mu}_Q(\mathbf{v}) - \boldsymbol{\mu}_P(\mathbf{v}))^2 \quad (2.305)$$

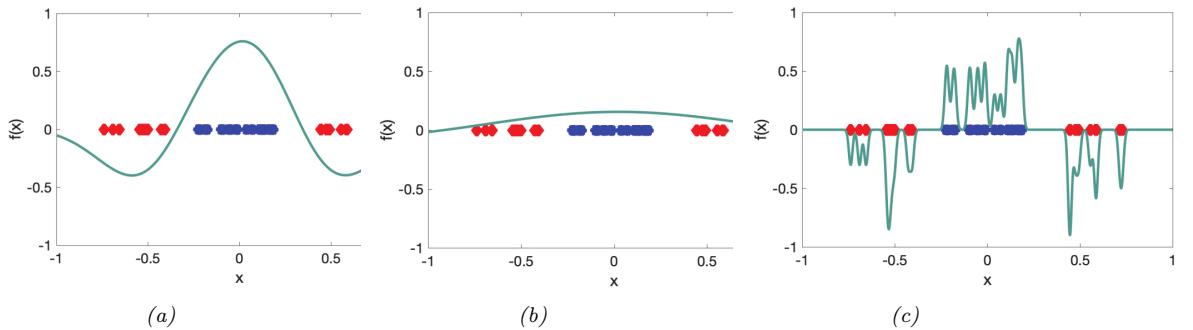


Figure 2.22: Effect of decreasing the bandwidth parameter σ on the witness function defined by a Gaussian kernel. From a figure from [GSJ19]. Used with kind permission of Dougal Sutherland.

at a set of test locations $\mathbf{v}_1, \dots, \mathbf{v}_J$ is enough to detect a difference between P and Q . Hence we define the (squared) UME as follows:

$$\text{UME}^2(P, Q) = \frac{1}{J} \sum_{j=1}^J [\mu_P(\mathbf{v}_j) - \mu_Q(\mathbf{v}_j)]^2 \quad (2.306)$$

where $\mu_P(\mathbf{v}) = \mathbb{E}_{p(\mathbf{x})} [\mathcal{K}(\mathbf{x}, \mathbf{v})]$ can be estimated empirically in $O(N)$ time, and similarly for $\mu_Q(\mathbf{v})$.

A normalized version of UME, known as NME, is presented in [Jit+16]. By maximizing NME wrt the locations \mathbf{v}_j , we can maximize the statistical power of the test, and find locations where P and Q differ the most. This provides an interpretable two-sample test for high dimensional data.

2.7.3.4 Choosing the right kernel

The effectiveness of MMD (and UME) obviously crucially depends on the right choice of kernel. Even for distinguishing 1d samples, the choice of kernel can be very important. For example, consider a Gaussian kernel, $\mathcal{K}_\sigma(\mathbf{x}, \mathbf{x}') = \exp(-\frac{1}{2\sigma^2} \|\mathbf{x} - \mathbf{x}'\|^2)$. The effect of changing σ in terms of the ability to distinguish two different sets of 1d samples is shown in Figure 2.22. Fortunately, the MMD is differentiable wrt the kernel parameters, so we can choose the optimal σ^2 so as to maximize the power of the test [Sut+17]. (See also [Fla+16] for a Bayesian approach, which maximizes the marginal likelihood of a GP representation of the kernel mean embedding.)

For high-dimensional data such as images, it can be useful to use a pre-trained CNN model as a way to compute low-dimensional features. For example, we can define $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_\sigma(\mathbf{h}(\mathbf{x}), \mathbf{h}(\mathbf{x}'))$, where \mathbf{h} is some hidden layer of a CNN, such as the ‘inception’ model of [Sze+15a]. The resulting MMD metric is known as the **kernel inception distance** [Biñ+18]. This is similar to the **Fréchet inception distance** [Heu+17a], but has nicer statistical properties, and is better correlated with human perceptual judgement [Zho+19a].

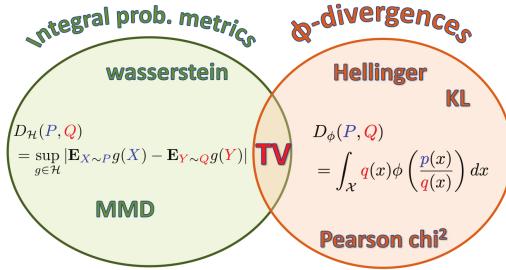


Figure 2.23: Summary of the two main kinds of divergence measures between two probability distributions P and Q . From a figure from [GSJ19]. Used with kind permission of Arthur Gretton.

2.7.4 Total variation distance

The **total variation distance** between two probability distributions is defined as follows:

$$D_{\text{TV}}(p, q) \triangleq \frac{1}{2} \|p - q\|_1 = \frac{1}{2} \int |p(\mathbf{x}) - q(\mathbf{x})| d\mathbf{x} \quad (2.307)$$

This is equal to an f -divergence where $f(r) = |r - 1|/2$, since

$$\frac{1}{2} \int q(\mathbf{x}) \left| \frac{p(\mathbf{x})}{q(\mathbf{x})} - 1 \right| d\mathbf{x} = \frac{1}{2} \int q(\mathbf{x}) \left| \frac{p(\mathbf{x}) - q(\mathbf{x})}{q(\mathbf{x})} \right| d\mathbf{x} = \frac{1}{2} \int |p(\mathbf{x}) - q(\mathbf{x})| d\mathbf{x} \quad (2.308)$$

One can also show that the TV distance is an integral probability measure. In fact, it is the only divergence that is both an IPM and an f -divergence [Sri+09]. See Figure 2.23 for a visual summary.

2.7.5 Density ratio estimation using binary classifiers

In this section, we discuss a simple approach for comparing two distributions that turns out to be equivalent to IPMs and f -divergences.

Consider a binary classification problem in which points from P have label $y = 1$ and points from Q have label $y = 0$, i.e., $P(\mathbf{x}) = p(\mathbf{x}|y = 1)$ and $Q(\mathbf{x}) = p(\mathbf{x}|y = 0)$. Let $p(y = 1) = \pi$ be the class prior. By Bayes' rule, the density ratio $r(\mathbf{x}) = P(\mathbf{x})/Q(\mathbf{x})$ is given by

$$\frac{P(\mathbf{x})}{Q(\mathbf{x})} = \frac{p(\mathbf{x}|y = 1)}{p(\mathbf{x}|y = 0)} = \frac{p(y = 1|\mathbf{x})p(\mathbf{x})}{p(y = 1)} / \frac{p(y = 0|\mathbf{x})p(\mathbf{x})}{p(y = 0)} \quad (2.309)$$

$$= \frac{p(y = 1|\mathbf{x})}{p(y = 0|\mathbf{x})} \frac{1 - \pi}{\pi} \quad (2.310)$$

If we assume $\pi = 0.5$, then we can estimate the ratio $r(\mathbf{x})$ by fitting a binary classifier or discriminator $h(\mathbf{x}) = p(y = 1|\mathbf{x})$ and then computing $r = h/(1 - h)$. This is called the **density ratio estimation** or **DRE** trick.

We can optimize the classifier h by minimizing the risk (expected loss). For example, if we use

log-loss, we have

$$R(h) = \mathbb{E}_{p(\mathbf{x}|y)p(y)} [-y \log h(\mathbf{x}) - (1-y) \log(1-h(\mathbf{x}))] \quad (2.311)$$

$$= \pi \mathbb{E}_{P(\mathbf{x})} [-\log h(\mathbf{x})] + (1-\pi) \mathbb{E}_{Q(\mathbf{x})} [-\log(1-h(\mathbf{x}))] \quad (2.312)$$

We can also use other loss functions $\ell(y, h(\mathbf{x}))$ (see Section 26.2.2).

Let $R_{h^*}^\ell = \inf_{h \in \mathcal{F}} R(h)$ be the minimum risk achievable for loss function ℓ , where we minimize over some function class \mathcal{F} .¹¹ In [NWJ09], they show that for every f -divergence, there is a loss function ℓ such that $-D_f(P, Q) = R_{h^*}^\ell$. For example (using the notation $\tilde{y} \in \{-1, 1\}$ instead of $y \in \{0, 1\}$), total-variation distance corresponds to hinge loss, $\ell(\tilde{y}, h) = \max(0, 1 - \tilde{y}h)$; Hellinger distance corresponds to exponential loss, $\ell(\tilde{y}, h) = \exp(-\tilde{y}h)$; and χ^2 divergence corresponds to logistic loss, $\ell(\tilde{y}, h) = \log(1 + \exp(-\tilde{y}h))$.

We can also establish a connection between binary classifiers and IPMs [Sri+09]. In particular, let $\ell(\tilde{y}, h) = -2\tilde{y}h$, and $p(\tilde{y} = 1) = p(\tilde{y} = -1) = 0.5$. Then we have

$$R_{h^*} = \inf_h \int \ell(\tilde{y}, h(\mathbf{x})) p(\mathbf{x}|\tilde{y}) p(\tilde{y}) d\mathbf{x} d\tilde{y} \quad (2.313)$$

$$= \inf_h 0.5 \int \ell(1, h(\mathbf{x})) p(\mathbf{x}|\tilde{y}=1) d\mathbf{x} + 0.5 \int \ell(-1, h(\mathbf{x})) p(\mathbf{x}|\tilde{y}=-1) d\mathbf{x} \quad (2.314)$$

$$= \inf_h \int h(\mathbf{x}) Q(\mathbf{x}) d\mathbf{x} - \int h(\mathbf{x}) P(\mathbf{x}) d\mathbf{x} \quad (2.315)$$

$$= \sup_h - \int h(\mathbf{x}) Q(\mathbf{x}) d\mathbf{x} + \int h(\mathbf{x}) P(\mathbf{x}) d\mathbf{x} \quad (2.316)$$

which matches Equation (2.295). Thus the classifier plays the same role as the witness function.

¹¹ If P is a fixed distribution, and we minimize the above objective wrt h , while also maximizing it wrt a model $Q(\mathbf{x})$, we recover a technique known as a generative adversarial network for fitting an implicit model to a distribution of samples P (see Chapter 26 for details). However, in this section, we assume Q is known.

3 Statistics

3.1 Introduction

Probability theory (which we discussed in Chapter 2) is concerned with modeling the distribution over observed data outcomes \mathcal{D} given known parameters $\boldsymbol{\theta}$ by computing $p(\mathcal{D}|\boldsymbol{\theta})$. By contrast, **statistics** is concerned with the inverse problem, in which we want to infer the unknown parameters $\boldsymbol{\theta}$ given observations, i.e., we want to compute $p(\boldsymbol{\theta}|\mathcal{D})$. Indeed, statistics was originally called **inverse probability theory**. Nowadays, there are two main approaches to statistics, **frequentist statistics** and **Bayesian statistics**, as we discuss below. (See also Section 34.1, where we compare the frequentist and Bayesian approaches to decision theory.) Note, however, that most of this chapter (and the entire book) focuses on the Bayesian approach, for reasons that will become clear.

3.2 Bayesian statistics

In the Bayesian approach to statistics, we treat the parameters $\boldsymbol{\theta}$ as unknown, and the data \mathcal{D} as fixed and known. (This is the opposite of the frequentist approach, which we discuss in Section 3.3.) We represent our uncertainty about the parameters, after (posterior to) seeing the data, by computing the **posterior distribution** using Bayes' rule:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})}{p(\mathcal{D})} = \frac{p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})}{\int p(\boldsymbol{\theta}')p(\mathcal{D}|\boldsymbol{\theta}')d\boldsymbol{\theta}'} \quad (3.1)$$

Here $p(\boldsymbol{\theta})$ is called the **prior**, and represents our beliefs about the parameters before seeing the data; $p(\mathcal{D}|\boldsymbol{\theta})$ is called the **likelihood**, and represents our beliefs about what data we expect to see for each setting of the parameters; $p(\boldsymbol{\theta}|\mathcal{D})$ is called the **posterior**, and represents our beliefs about the parameters after seeing the data; and $p(\mathcal{D})$ is called the **marginal likelihood** or **evidence**, and is a normalization constant that we will use later.

The task of computing this posterior is called **Bayesian inference**, **posterior inference**, or just **inference**. We will give many examples in the following sections of this chapter, and will discuss algorithmic issues in Part II. For more details on Bayesian statistics, see e.g., [Ber97a; Hof09; Lam18; Kru15; McE20] for introductory level material, [Gel+14a; MKL21; GHV20a] for intermediate level material, and [BS94; Ber85b; Rob07] for more advanced theory.

3.2.1 Tossing coins

It is common to explain the key ideas behind Bayesian inference by considering a coin tossing experiment. We shall follow this tradition (although also see [Supplementary](#) Section 3.1 for an alternative gentle introduction to Bayes using the example of Bayesian concept learning.)

Let $\theta \in [0, 1]$ be the chance that some coin comes up heads, an event we denote by $Y = 1$. Suppose we toss a coin N times, and we record the outcomes as $\mathcal{D} = \{y_n \in \{0, 1\} : n = 1 : N\}$. We want to compute $p(\theta|\mathcal{D})$, which represents our beliefs about the parameter after doing collecting the data. To compute the posterior, we can use Bayes' rule, as in Equation (3.1). We give the details below.

3.2.1.1 Likelihood

We assume the data are **iid** or **independent and identically distributed**. Thus the likelihood has the form

$$p(\mathcal{D}|\theta) = \prod_{n=1}^N \theta^{y_n} (1 - \theta)^{1-y_n} = \theta^{N_1} (1 - \theta)^{N_0} \quad (3.2)$$

where we have defined $N_1 = \sum_{n=1}^N \mathbb{I}(y_n = 1)$ and $N_0 = \sum_{n=1}^N \mathbb{I}(y_n = 0)$, representing the number of heads and tails. These counts are called the **sufficient statistics** of the data, since this is all we need to know about \mathcal{D} to infer θ . The total count, $N = N_0 + N_1$, is called the **sample size**.

Note that we can also consider a Binomial likelihood model, in which we perform N trials and observe the number of heads, y , rather than observing a sequence of coin tosses. Now the likelihood has the following form:

$$p(\mathcal{D}|\theta) = \text{Bin}(y|N, \theta) = \binom{N}{y} \theta^y (1 - \theta)^{N-y} \quad (3.3)$$

The scaling factor $\binom{N}{y}$ is independent of θ , so we can ignore it. Thus this likelihood is proportional to the Bernoulli likelihood in Equation (3.2), so our inferences about θ will be the same for both models.

3.2.1.2 Prior

We also need to specify a prior. Let us assume we know nothing about the parameter, except that it lies in the interval $[0, 1]$. We can represent this **uninformative prior** using a uniform distribution,

$$p(\theta) = \text{Unif}(\theta|0, 1) \quad (3.4)$$

More generally, we will write the prior using a **beta distribution** (Section 2.2.4.1), for reasons that will become clear shortly. That is, we assume

$$p(\theta) = \text{Beta}(\theta|\check{\alpha}, \check{\beta}) \propto \theta^{\check{\alpha}-1} (1 - \theta)^{\check{\beta}-1} \quad (3.5)$$

Here $\check{\alpha}$ and $\check{\beta}$ are called **hyper-parameters**, since they are parameters of the prior which determine our beliefs about the “main” parameter θ . If we set $\check{\alpha}=\check{\beta}=1$, we recover the uniform prior as a special case.

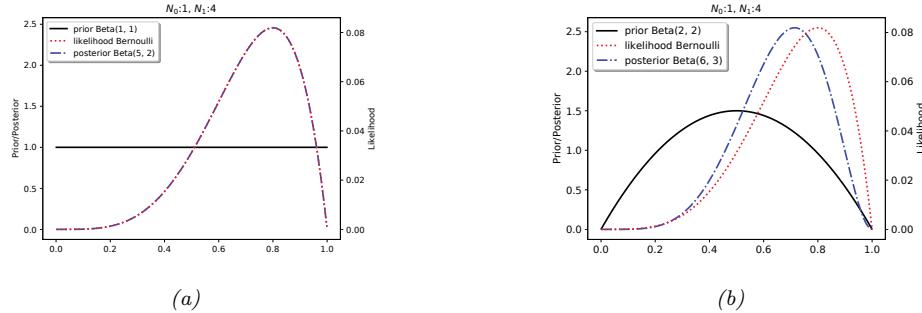


Figure 3.1: Updating a Beta prior with a Bernoulli likelihood with sufficient statistics $N_1 = 4, N_0 = 1$. (a) Uniform Beta(1,1) prior. (a) Beta(2,2) prior. Generated by `beta_binom_post_plot.ipynb`.

We can think of these hyper-parameters as **pseudocounts**, which play a role analogous to the empirical counts N_1 and N_0 derived from the real data. The strength of the prior is controlled by $\check{N} = \check{\alpha} + \check{\beta}$; this is called the **equivalent sample size**, since it plays a role analogous to the observed sample size, $N = N_1 + N_0$.

3.2.1.3 Posterior

We can compute the posterior by multiplying the likelihood by the prior:

$$p(\theta|\mathcal{D}) \propto \theta^{N_1}(1-\theta)^{N_0} \theta^{\check{\alpha}-1}(1-\theta)^{\check{\beta}-1} \propto \text{Beta}(\theta|\check{\alpha}+N_1, \check{\beta}+N_0) = \text{Beta}(\theta|\hat{\alpha}, \hat{\beta}) \quad (3.6)$$

where $\hat{\alpha} \triangleq \check{\alpha} + N_1$ and $\hat{\beta} \triangleq \check{\beta} + N_0$ are the parameters of the posterior. Since the posterior has the same functional form as the prior, we say that it is a **conjugate prior** (see Section 3.4 for more details).

For example, suppose we observe $N_1 = 4$ heads and $N_0 = 1$ tails. If we use a uniform prior, we get the posterior shown in Figure 3.1a. Not surprisingly, this has exactly the same shape as the likelihood (but is scaled to integrate to 1 over the range $[0, 1]$).

Now suppose we use a prior that has a slight preference for values of θ near to 0.5, reflecting our prior belief that it is more likely than not that the coin is fair. We will make this a weak prior by setting $\check{\alpha} = \check{\beta} = 2$. The effect of using this prior is illustrated in Figure 3.1b. We see the posterior (blue line) is a “compromise” between the prior (red line) and the likelihood (black line).

3.2.1.4 Posterior mode (MAP estimate)

The most probable value of the parameter is given by the MAP estimate

$$\hat{\theta}_{\text{map}} = \arg \max_{\theta} p(\theta|\mathcal{D}) = \arg \max_{\theta} \log p(\theta|\mathcal{D}) = \arg \max_{\theta} \log p(\theta) + \log p(\mathcal{D}|\theta) \quad (3.7)$$

Using calculus, one can show that this is given by

$$\hat{\theta}_{\text{map}} = \frac{\check{\alpha} + N_1 - 1}{\check{\alpha} + N_1 - 1 + \check{\beta} + N_0 - 1} \quad (3.8)$$

If we use a uniform prior, $p(\theta) \propto 1$, the MAP estimate becomes the MLE, since $\log p(\theta) = 0$:

$$\hat{\theta}_{\text{mle}} = \arg \max_{\theta} \log p(\mathcal{D}|\theta) = \frac{N_1}{N_1 + N_0} = \frac{N_1}{N} \quad (3.9)$$

This is intuitive and easy to compute. However, the MLE can be very misleading in the small sample setting. For example, suppose we toss the coins N times, but never see any heads, so $N_1 = 0$. In this case, we would estimate that $\hat{\theta} = 0$, which means we would not predict any future observations to be heads either. This is a very extreme estimate, that is likely due to insufficient data. We can solve this problem using a MAP estimate with a stronger prior. For example, if we use a $\text{Beta}(\theta|2, 2)$ prior, we get the estimate

$$\hat{\theta}_{\text{map}} = \frac{N_1 + 1}{N_1 + 1 + N_0 + 1} = \frac{N_1 + 1}{N + 2} \quad (3.10)$$

This is called **add-one smoothing**.

3.2.1.5 Posterior mean

The posterior mode can be a poor summary of the posterior, since it corresponds to picking a single point from the entire distribution. The posterior mean is a more robust estimate, since it is a summary statistic derived by integrating over the distribution, $\bar{\theta} = \int \theta p(\theta|\mathcal{D})d\theta$. In the case of a beta posterior, $p(\theta|\mathcal{D}) = \text{Beta}(\theta|\bar{\alpha}, \bar{\beta})$, the posterior mean is given by

$$\bar{\theta} \triangleq \mathbb{E}[\theta|\mathcal{D}] = \frac{\bar{\alpha}}{\bar{\beta} + \bar{\alpha}} = \frac{\bar{\alpha}}{\bar{N}} \quad (3.11)$$

where $\bar{N} = \bar{\beta} + \bar{\alpha}$ is the strength (equivalent sample size) of the posterior.

We will now show that the posterior mean is a convex combination of the prior mean, $m = \check{\alpha} / \check{N}$ and the MLE, $\hat{\theta}_{\text{mle}} = \frac{N_1}{N}$:

$$\mathbb{E}[\theta|\mathcal{D}] = \frac{\check{\alpha} + N_1}{\check{\alpha} + N_1 + \check{\beta} + N_0} = \frac{\check{N} m + N_1}{\check{N} + \check{N}} \quad (3.12)$$

$$= \frac{\check{N}}{\check{N} + \check{N}} m + \frac{N}{\check{N} + \check{N}} \frac{N_1}{N} = \lambda m + (1 - \lambda) \hat{\theta}_{\text{mle}} \quad (3.13)$$

where $\lambda = \frac{\check{N}}{\check{N}}$ is the ratio of the prior to posterior equivalent sample size. We see that the weaker the prior is, the smaller λ is, and hence the closer the posterior mean is to the MLE.

3.2.1.6 Posterior variance

To capture some notion of uncertainty in our estimate, a common approach is to compute the **standard error** of our estimate, which is just the posterior standard deviation:

$$\text{se}(\theta) = \sqrt{\mathbb{V}[\theta|\mathcal{D}]} \quad (3.14)$$

In the case of the Bernoulli model, we showed that the posterior is a beta distribution. The variance of the beta posterior is given by

$$\mathbb{V}[\theta|\mathcal{D}] = \frac{\widehat{\alpha}\widehat{\beta}}{(\widehat{\alpha} + \widehat{\beta})^2(\widehat{\alpha} + \widehat{\beta} + 1)} = \frac{(\check{\alpha} + N_1)(\check{\beta} + N_0)}{(\check{\alpha} + N_1 + \check{\beta} + N_0)^2(\check{\alpha} + N_1 + \check{\beta} + N_0 + 1)} \quad (3.15)$$

If $N \gg \check{\alpha} + \check{\beta}$, this simplifies to

$$\mathbb{V}[\theta|\mathcal{D}] \approx \frac{N_1 N_0}{(N_1 + N_0)^2(N_1 + N_0)} = \frac{N_1}{N} \frac{N_0}{N} \frac{1}{N} = \frac{\hat{\theta}(1 - \hat{\theta})}{N} \quad (3.16)$$

where $\hat{\theta} = N_1/N$ is the MLE. Hence the standard error is given by

$$\sigma = \sqrt{\mathbb{V}[\theta|\mathcal{D}]} \approx \sqrt{\frac{\hat{\theta}(1 - \hat{\theta})}{N}} \quad (3.17)$$

We see that the uncertainty goes down at a rate of $1/\sqrt{N}$. We also see that the uncertainty (variance) is maximized when $\hat{\theta} = 0.5$, and is minimized when $\hat{\theta}$ is close to 0 or 1. This makes sense, since it is easier to be sure that a coin is biased than to be sure that it is fair.

3.2.1.7 Credible intervals

A posterior distribution is (usually) a high dimensional object that is hard to visualize and work with. A common way to summarize such a distribution is to compute a point estimate, such as the posterior mean or mode, and then to compute a **credible interval**, which quantifies the uncertainty associated with that estimate. (A credible interval is not the same as a confidence interval, which is a concept from frequentist statistics which we discuss in Section 3.3.5.1.)

More precisely, we define a $100(1 - \alpha)\%$ credible interval to be a (contiguous) region $C = (\ell, u)$ (standing for lower and upper) which contains $1 - \alpha$ of the posterior probability mass, i.e.,

$$C_\alpha(\mathcal{D}) = (\ell, u) : P(\ell \leq \theta \leq u|\mathcal{D}) = 1 - \alpha \quad (3.18)$$

There may be many intervals that satisfy Equation (3.18), so we usually choose one such that there is $(1 - \alpha)/2$ mass in each tail; this is called a **central interval**. If the posterior has a known functional form, we can compute the posterior central interval using $\ell = F^{-1}(\alpha/2)$ and $u = F^{-1}(1 - \alpha/2)$, where F is the cdf of the posterior, and F^{-1} is the inverse cdf. For example, if the posterior is Gaussian, $p(\theta|\mathcal{D}) = \mathcal{N}(0, 1)$, and $\alpha = 0.05$, then we have $\ell = \Phi^{-1}(\alpha/2) = -1.96$, and $u = \Phi^{-1}(1 - \alpha/2) = 1.96$, where Φ denotes the cdf of the Gaussian. This justifies the common practice of quoting a credible interval in the form of $\mu \pm 2\sigma$, where μ represents the posterior mean, σ represents the posterior standard deviation, and 2 is a good approximation to 1.96.

A problem with central intervals is that there might be points outside the central interval which have higher probability than points that are inside, as illustrated in Figure 3.2(a). This motivates an alternative quantity known as the **highest posterior density** or **HPD** region, which is the set of points which have a probability above some threshold. More precisely we find the threshold p^* on the pdf such that

$$1 - \alpha = \int_{\theta:p(\theta|\mathcal{D})>p^*} p(\theta|\mathcal{D}) d\theta \quad (3.19)$$

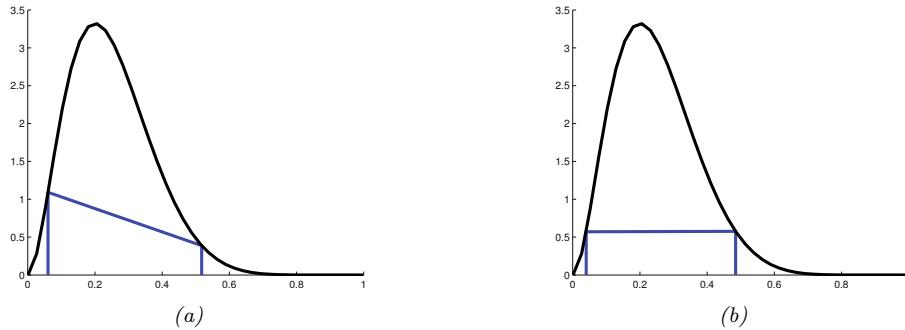


Figure 3.2: (a) Central interval and (b) HPD region for a Beta(3,9) posterior. The CI is (0.06, 0.52) and the HPD is (0.04, 0.48). Adapted from Figure 3.6 of [Hof09]. Generated by [betaHPD.ipynb](#).

and then define the HPD as

$$C_\alpha(\mathcal{D}) = \{\theta : p(\theta|\mathcal{D}) \geq p^*\} \quad (3.20)$$

In 1d, the HPD region is sometimes called a **highest density interval** or **HDI**. For example, Figure 3.2(b) shows the 95% HDI of a Beta(3,9) distribution, which is (0.04, 0.48). We see that this is narrower than the central interval, even though it still contains 95% of the mass; furthermore, every point inside of it has higher density than every point outside of it.

3.2.1.8 Posterior predictive distribution

Suppose we want to predict future observations. The optimal Bayesian approach is to compute the **posterior predictive distribution**, by marginalizing out all the unknown parameters:

$$p(\mathbf{y}|\mathcal{D}) = \int p(\mathbf{y}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (3.21)$$

Sometimes computing this integral can be difficult (even if we already have access to the posterior). A common approximation is to just “**plug in**” a point estimate of the parameters, $\hat{\boldsymbol{\theta}} = \delta(\mathcal{D})$, where $\delta()$ is some estimator such as a method to compute the MLE or MAP, which gives

$$p(\mathbf{y}|\mathcal{D}) \approx p(\mathbf{y}|\hat{\boldsymbol{\theta}}) \quad (3.22)$$

This is called a **plugin approximation**. This is equivalent to modeling the posterior with a degenerate distribution centered at the point estimate

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \quad (3.23)$$

where δ is the Dirac delta function. This follows from the **sifting property** of delta functions:

$$p(\mathbf{y}|\mathcal{D}) = \int p(\mathbf{y}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} = \int p(\mathbf{y}|\boldsymbol{\theta})\delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})d\boldsymbol{\theta} = p(\mathbf{y}|\hat{\boldsymbol{\theta}}) \quad (3.24)$$

Unfortunately, the plugin approximation can result in overfitting. For example, consider the coin tossing example, and suppose we have seen $N = 3$ heads in a row. The MLE is $\hat{\theta} = 3/3 = 1$. However, if we use this estimate for prediction, we would predict that tails are impossible, and would be very surprised if one ever showed up.¹

Instead of the plugin approximation, we can marginalize over all parameter values to compute the exact posterior predictive, as follows:

$$p(y = 1|\mathcal{D}) = \int_0^1 p(y = 1|\theta)p(\theta|\mathcal{D})d\theta \quad (3.25)$$

$$= \int_0^1 \theta \text{Beta}(\theta|\widehat{\alpha}, \widehat{\beta})d\theta = \mathbb{E}[\theta|\mathcal{D}] = \frac{\widehat{\alpha}}{\widehat{\alpha} + \widehat{\beta}} \quad (3.26)$$

If we use a uniform prior, $p(\theta) = \text{Beta}(\theta|1, 1)$, the predictive distribution becomes

$$p(y = 1|\mathcal{D}) = \frac{N_1 + 1}{N_1 + N_0 + 2} \quad (3.27)$$

This is known as **Laplace's rule of succession**. Note that this is equivalent to plugging in the add-one smoothing estimate from Equation (3.10); however, that relied on the rather unnatural Beta(2,2) prior, whereas Laplace smoothing uses a uniform prior.

3.2.1.9 Marginal likelihood

The **marginal likelihood** or **evidence** for a model \mathcal{M} is defined as

$$p(\mathcal{D}|\mathcal{M}) = \int p(\theta|\mathcal{M})p(\mathcal{D}|\theta, \mathcal{M})d\theta \quad (3.28)$$

When performing inference for the parameters of a specific model, we can ignore this term, since it is constant wrt θ . However, this quantity plays a vital role when choosing between different models, as we discuss in Section 3.8.1. It is also useful for estimating the hyperparameters from data (an approach known as empirical Bayes), as we discuss in Section 3.7.

In general, computing the marginal likelihood can be hard. However, in the case of the beta-Bernoulli model, the marginal likelihood is proportional to the ratio of the posterior normalizer to the prior normalizer. To see this, recall that the posterior is given by $p(\theta|\mathcal{D}) = \text{Beta}(\theta|\widehat{\alpha}, \widehat{\beta})$, where $\widehat{\alpha}=\check{\alpha}+N_1$ and $\widehat{\beta}=\check{\beta}+N_0$. We know the normalization constant of the posterior is $B(\widehat{\alpha}, \widehat{\beta})$, where B is the beta function. Hence

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})} = \frac{1}{p(\mathcal{D})} \left[\theta^{N_1} (1-\theta)^{N_0} \right] \left[\frac{1}{B(\check{\alpha}, \check{\beta})} \theta^{\check{\alpha}-1} (1-\theta)^{\check{\beta}-1} \right] \quad (3.29)$$

$$= \frac{1}{p(\mathcal{D})} \frac{1}{B(\check{\alpha}, \check{\beta})} \left[\theta^{\check{\alpha}+N_1-1} (1-\theta)^{\check{\beta}+N_0-1} \right] = \frac{1}{B(\widehat{\alpha}, \widehat{\beta})} \left[\theta^{\widehat{\alpha}} (1-\theta)^{\widehat{\beta}} \right] \quad (3.30)$$

So the marginal likelihood is given by the ratio of normalization constants for the posterior and prior:

$$p(\mathcal{D}) = \frac{B(\widehat{\alpha}, \widehat{\beta})}{B(\check{\alpha}, \check{\beta})} \quad (3.31)$$

1. This is analogous to a **black swan event**, which refers to the discovery of black swans by Dutch explorers when they first arrived in Australia in 1697, after only ever having seen white swans their entire lives (see https://en.wikipedia.org/wiki/Black_swan_theory for details).

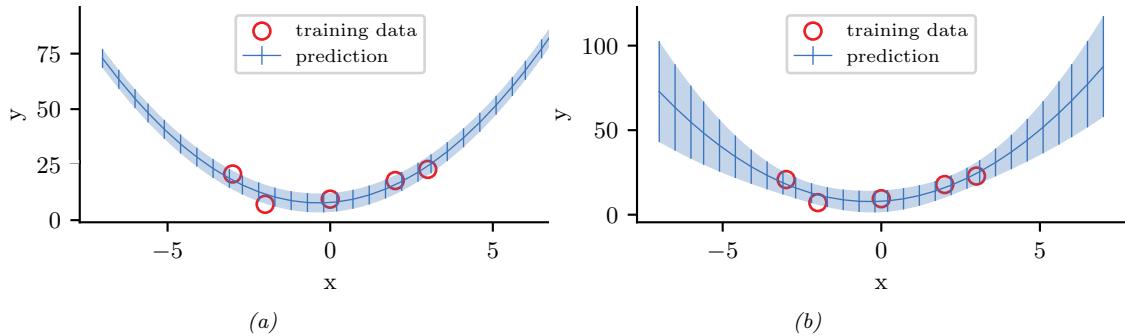


Figure 3.3: Predictions made by a polynomial regression model fit to a small dataset. (a) Plugin approximation to predictive density using the MLE. The curves shows the posterior mean, $\mathbb{E}[y|\mathbf{x}]$, and the error bars show the posterior standard deviation, $\text{std}[y|\mathbf{x}]$, around this mean. (b) Bayesian posterior predictive density, obtained by integrating out the parameters. Generated by [linreg_post_pred_plot.ipynb](#).

3.2.2 Modeling more complex data

In Section 3.2.1, we showed how the Bayesian approach can be applied to analyse a very simple model, namely a Bernoulli distribution for representing binary events such as coin tosses. The same basic ideas can be applied to more complex models. For example, in machine learning, we are often very interested in predicting outcomes \mathbf{y} given input features \mathbf{x} . For this, we can use a conditional probability distribution of the form $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$, which can be a generalized linear model (Chapter 15), or a neural network (Chapter 16), etc.

The main quantity of interest is the **posterior predictive distribution**, given by

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (3.32)$$

By **integrating out**, or **marginalizing out**, the unknown parameters, we reduce the chance of overfitting, since we are effectively computing the weighted average of predictions from an infinite number of models. This act of integrating over uncertainty is at the heart of the Bayesian approach to machine learning. (Of course, the Bayesian approach requires a prior, but so too do methods that rely on regularization, so the prior is not so much the distinguishing aspect.)

It is worth contrasting the Bayesian approach to the more common **plugin approximation**, in which we compute a point estimate $\hat{\boldsymbol{\theta}}$ of the parameters (such as the MLE), and then plug them into the model to make predictions using $p(\mathbf{y}|\mathbf{x}, \hat{\boldsymbol{\theta}})$. As we explained in Section 3.2.1.8, this is equivalent to approximate the posterior by a delta function, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})$, since

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})\delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})d\boldsymbol{\theta} = p(\mathbf{y}|\mathbf{x}, \hat{\boldsymbol{\theta}}) \quad (3.33)$$

The plugin approximation is simple and widely used. However, it ignores uncertainty in the parameter estimates, which can result in an underestimate of the predictive uncertainty. For example, Figure 3.3a plots the plugin approximation $p(\mathbf{y}|\mathbf{x}, \hat{\boldsymbol{\theta}})$ for a linear regression model $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\hat{\mathbf{w}}_{\text{mle}}^T \mathbf{x}, \hat{\sigma}_{\text{mle}}^2)$, where we plug in the MLEs for \mathbf{w} and σ^2 . (See Section 15.2.1 for details on how to compute these MLEs.) We see that the size of the predicted variance is a constant (namely $\hat{\sigma}^2$).

The uncertainty captured by σ is called **aleatoric uncertainty** or **intrinsic uncertainty**, and would persist even if we knew the true model and true parameters. However, since we don't know the parameters, we have an additional, and orthogonal, source of uncertainty, called **epistemic uncertainty** (since it arises due to a lack of knowledge about the truth). In the Bayesian approach, we take this into account, which can be useful for applications such as active learning (Section 34.7), Bayesian optimization (Section 6.6), and risk-sensitive decision making (Section 34.1.3). The resulting Bayesian posterior predictive distribution for this example is shown in Figure 3.3b. We see that now the error bars get wider as we move away from the training data. For more details on Bayesian linear regression, see Section 15.2.

We can use similar Bayesian methods for more complex nonlinear models such as neural nets, as we discuss in Section 17.1, as well as for unconditional generative models, as we discuss in Part IV.

3.2.3 Selecting the prior

A challenge with the Bayesian approach is that it requires the user to specify a prior, which may be difficult in large models, such as neural networks. We discuss the topic of prior selection at length later in this chapter. In particular, in Section 3.4, we discuss **conjugate priors**, which are computationally convenient; in Section 3.5, we discuss **uninformative priors**, which often correspond to a limit of a conjugate prior where we “know nothing”; in Section 3.6, we discuss **hierarchical priors**, which are useful when we have multiple related datasets; and in Section 3.7, we discuss **empirical priors**, which can be learned from the data.

3.2.4 Computational issues

Another challenge with the Bayesian approach is that it can be computationally expensive to compute the posterior and/or posterior predictive. We give an overview of suitable approximate posterior inference methods in Section 7.4, and discuss the topic at length in Part II. (See also [MFR20] for a historical review of this topic.)

3.2.5 Exchangeability and de Finetti's theorem

An interesting philosophical question is: where do priors come from, given that they refer to parameters which are just abstract quantities in a model, and not directly observable. A fundamental result, known as **de Finetti's theorem**, explains how they are related to our beliefs about observable outcomes.

To explain the result, we first need a definition. We say that a sequence of random variables $(\mathbf{x}_1, \mathbf{x}_2, \dots)$ is **infinitely exchangeable** if, for any n , the joint probability $p(\mathbf{x}_1, \dots, \mathbf{x}_n)$ is invariant to permutation of the indices. That is, for any permutation π , we have

$$p(\mathbf{x}_1, \dots, \mathbf{x}_n) = p(\mathbf{x}_{\pi_1}, \dots, \mathbf{x}_{\pi_n}) \quad (3.34)$$

Exchangeability is a more general concept compared to the more familiar concept of a sequence of independent, identically distributed or **iid** variables. For example, suppose $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ is a sequence of images, where each $\mathbf{x}_i \sim p^*$ is generated independently from the same “true distribution” p^* . We see that this is an iid sequence. Now suppose \mathbf{x}_0 is a background image. The sequence $(\mathbf{x}_0 + \mathbf{x}_1, \dots, \mathbf{x}_0 + \mathbf{x}_n)$ is infinitely exchangeable but not iid, since all the variables share a hidden

common factor, namely the background \mathbf{x}_0 . Thus the more examples we see, the better we will be able to estimate the shared \mathbf{x}_0 , and thus the better we can predict future elements.

More generally, we can view an exchangeable sequence as coming from a hidden common cause, which we can treat as an unknown random variable $\boldsymbol{\theta}$. This is formalized by **de Finetti's theorem**:

Theorem 3.2.1 (de Finetti's theorem). *A sequence of random variables $(\mathbf{x}_1, \mathbf{x}_2, \dots)$ is infinitely exchangeable iff, for all n , we have*

$$p(\mathbf{x}_1, \dots, \mathbf{x}_n) = \int \prod_{i=1}^n p(\mathbf{x}_i | \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \quad (3.35)$$

where $\boldsymbol{\theta}$ is some hidden common random variable (possibly infinite dimensional). That is, \mathbf{x}_i are iid conditional on $\boldsymbol{\theta}$.

We often interpret $\boldsymbol{\theta}$ as a parameter. The theorem tells us that, if our data is exchangeable, then there must exist a parameter $\boldsymbol{\theta}$, and a likelihood $p(\mathbf{x}_i | \boldsymbol{\theta})$, and a prior $p(\boldsymbol{\theta})$. Thus the Bayesian approach follows automatically from exchangeability [O'N09]. (The approach can also be extended to conditional probability models using a concept called **partially exchangeable** [Dia88a].)

3.3 Frequentist statistics

Bayesian statistics, which we discussed in Section 3.2, treats parameters of models just like any other unknown random variable, and applies the rules of probability theory to infer them from data. Attempts have been made to devise approaches to statistical inference that avoid treating parameters like random variables, and which thus avoid the use of priors and Bayes rule. This alternative approach is known as **frequentist statistics**, **classical statistics** or **orthodox statistics**.

The basic idea (formalized in Section 3.3.1) is to represent uncertainty by calculating how a quantity estimated from data (such as a parameter or a predicted label) would change if the data were changed. It is this notion of variation across repeated trials that forms the basis for modeling uncertainty used by the frequentist approach. By contrast, the Bayesian approach views probability in terms of information rather than repeated trials. This allows the Bayesian to compute the probability of one-off events, such as the probability that the polar ice cap will melt by 2030. In addition, the Bayesian approach avoids certain paradoxes that plague the frequentist approach (see Section 3.3.5), and which are a source of much confusion.

Despite the disadvantages of frequentist statistics, it is a widely used approach, and it has some concepts (such as cross validation, model checking and conformal prediction) that are useful even for Bayesians [Rub84]. Thus it is important to know some of the basic principles. We give a brief summary below of these principles below. For more details, see other textbooks, such as [Was04; Cox06; YS10; EH16].

3.3.1 Sampling distributions

In frequentist statistics, uncertainty is not represented by the posterior distribution of a random variable, but instead by the sampling distribution of an estimator. (We define these two terms below.)

As explained in the section on decision theory in Section 34.1.2, an **estimator** is a decision procedure that specifies what action to take given some observed data. In the context of parameter

estimation, where the action space is to return a parameter vector, we will denote this by $\hat{\boldsymbol{\theta}} = \hat{\Theta}(\mathcal{D})$. For example, $\hat{\boldsymbol{\theta}}$ could be the maximum likelihood estimate, the MAP estimate, or the method of moments estimate.

The **sampling distribution** of an estimator is the distribution of results we would see if we applied the estimator multiple times to different datasets sampled from some distribution; in the context of parameter estimation, it is the distribution of $\hat{\boldsymbol{\theta}}$, viewed as a random variable that depends on the random sample \mathcal{D} . In more detail, imagine sampling S different data sets, each of size N , from some true model $p(\mathbf{x}|\boldsymbol{\theta}^*)$ to generate

$$\tilde{\mathcal{D}}^{(s)} = \{\mathbf{x}_n \sim p(\mathbf{x}_n|\boldsymbol{\theta}^*) : n = 1 : N\} \quad (3.36)$$

We denote this by $\tilde{\mathcal{D}}^{(s)} \sim \boldsymbol{\theta}^*$ for brevity. Now apply the estimator to each $\tilde{\mathcal{D}}^{(s)}$ to get a set of estimates, $\{\hat{\boldsymbol{\theta}}(\tilde{\mathcal{D}}^{(s)})\}$. As we let $S \rightarrow \infty$, the distribution induced by this set is the sampling distribution of the estimator. More precisely, we have

$$p(\hat{\Theta}(\tilde{\mathcal{D}}) = \boldsymbol{\theta} | \tilde{\mathcal{D}} \sim \boldsymbol{\theta}^*) \approx \frac{1}{S} \sum_{s=1}^S \delta(\boldsymbol{\theta} - \hat{\Theta}(\tilde{\mathcal{D}}^{(s)})) \quad (3.37)$$

We often approximate this by Monte Carlo, as we discuss in Section 3.3.2, although in some cases we can compute approximate it analytically, as we discuss in Section 3.3.3.

3.3.2 Bootstrap approximation of the sampling distribution

In cases where the estimator is a complex function of the data, or when the sample size is small, it is often useful to approximate its sampling distribution using a Monte Carlo technique known as the **bootstrap** [ET93].

The idea is simple. If we knew the true parameters $\boldsymbol{\theta}^*$, we could generate many (say S) fake datasets, each of size N , from the true distribution, using $\tilde{\mathcal{D}}^{(s)} = \{\mathbf{x}_n \sim p(\mathbf{x}_n|\boldsymbol{\theta}^*) : n = 1 : N\}$. We could then compute our estimate from each sample, $\hat{\boldsymbol{\theta}}^s = \hat{\Theta}(\tilde{\mathcal{D}}^{(s)})$ and use the empirical distribution of the resulting $\hat{\boldsymbol{\theta}}^s$ as our estimate of the sampling distribution, as in Equation (3.37). Since $\boldsymbol{\theta}^*$ is unknown, the idea of the **parametric bootstrap** is to generate each sampled dataset using $\hat{\boldsymbol{\theta}} = \hat{\Theta}(\mathcal{D})$ instead of $\boldsymbol{\theta}^*$, i.e., we use $\tilde{\mathcal{D}}^{(s)} = \{\mathbf{x}_n \sim p(\mathbf{x}_n|\hat{\boldsymbol{\theta}}) : n = 1 : N\}$ in Equation (3.37). This is a plug-in approximation to the sampling distribution.

The above approach requires that we have a parametric generative model for the data, $p(\mathbf{x}|\boldsymbol{\theta})$. An alternative, called the **non-parametric bootstrap**, is to sample N data points from the original dataset with replacement. This creates a new distribution $\mathcal{D}^{(s)}$ which has the same size as the original. However, the number of unique data points in a bootstrap sample is just $0.632 \times N$, on average. (To see this, note that the probability an item is picked at least once is $(1 - (1 - 1/N)^N)$, which approaches $1 - e^{-1} \approx 0.632$ for large N .) Fortunately, various improved versions of the bootstrap have been developed (see e.g., [ET93]).

Figure 3.4(a-b) shows an example where we compute the sampling distribution of the MLE for a Bernoulli using the parametric bootstrap. (Results using the non-parametric bootstrap are essentially the same.) When $N = 10$, we see that the sampling distribution is asymmetric, and therefore quite far from Gaussian, but when $N = 100$, the distribution looks more Gaussian, as theory suggests (see Section 3.3.3).

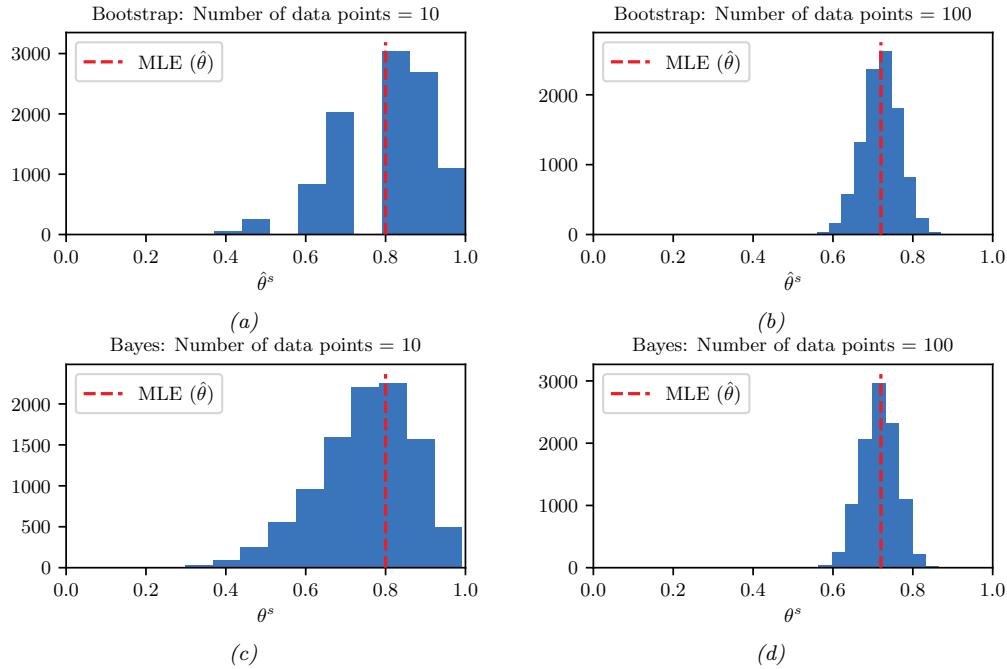


Figure 3.4: Bootstrap (top row) vs Bayes (bottom row). The N data cases were generated from $\text{Ber}(\theta = 0.7)$. Left column: $N = 10$. Right column: $N = 100$. (a-b) A bootstrap approximation to the sampling distribution of the MLE for a Bernoulli distribution. We show the histogram derived from $B = 10,000$ bootstrap samples. (c-d) Histogram of 10,000 samples from the posterior distribution using a uniform prior. Generated by [bootstrap_demo_bernoulli.ipynb](#).

A natural question is: what is the connection between the parameter estimates $\hat{\theta}^s = \hat{\Theta}(\mathcal{D}^{(s)})$ computed by the bootstrap and parameter values sampled from the posterior, $\theta^s \sim p(\cdot|\mathcal{D})$? Conceptually they are quite different. But in the common case that the estimator is MLE and the prior is not very strong, they can be quite similar. For example, Figure 3.4(c-d) shows an example where we compute the posterior using a uniform Beta(1,1) prior, and then sample from it. We see that the posterior and the sampling distribution are quite similar. So one can think of the bootstrap distribution as a “poor man’s” posterior [HTF01, p235].

However, perhaps surprisingly, bootstrap can be slower than posterior sampling. The reason is that the bootstrap has to generate S sampled datasets, and then fit a model to each one. By contrast, in posterior sampling, we only have to “fit” a model once given a single dataset. (Some methods for speeding up the bootstrap when applied to massive data sets are discussed in [Kle+11].)

3.3.3 Asymptotic normality of the sampling distribution of the MLE

The most common estimator is the MLE. When the sample size becomes large, the sampling distribution of the MLE for certain models becomes Gaussian. This is known as the **asymptotic normality** of the sampling distribution. More formally, we have the following result:

Theorem 3.3.1. *Under various technical conditions, we have*

$$\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \rightarrow \mathcal{N}(\mathbf{0}, \mathbf{F}(\boldsymbol{\theta}^*)^{-1}) \quad (3.38)$$

where $\mathbf{F}(\boldsymbol{\theta}^*)$ is the Fisher information matrix, defined in Equation (3.40), $\boldsymbol{\theta}^*$ are the parameters of the data generating process to which the estimator will be applied, and \rightarrow means convergence in distribution.

The Fisher information matrix equals the Hessian of the log likelihood, as we show in Section 3.3.4, so $\mathbf{F}(\boldsymbol{\theta}^*)$ measures the amount of curvature of the log-likelihood surface at the true parameter value. Thus we can interpret this theorem as follows: as the sample size goes to infinity, the sampling distribution of the MLE will converge to a Gaussian centered on the true parameter, with a precision equal to the Fisher information. Thus a problem with an informative (peaked) likelihood will ensure that the parameters are “well determined” by the data, and hence there will be little variation in the estimates $\hat{\boldsymbol{\theta}}$ around $\boldsymbol{\theta}^*$ as this estimator is applied across different datasets $\tilde{\mathcal{D}}$.

3.3.4 Fisher information matrix

In this section, we discuss an important quantity called the **Fisher information matrix**, which is related to the curvature of the log likelihood function. This plays a key role in frequentist statistics, for characterizing the sampling distribution of the MLE, discussed in Section 3.3.3. However, it is also used in Bayesian statistics (to derive Jeffreys’ uninformative priors, discussed in Section 3.5.2), as well as in optimization (as part of the natural gradient descent, procedure, discussed in Section 6.4).

3.3.4.1 Definition

The **score function** is defined to be the gradient of the log likelihood wrt the parameter vector:

$$\mathbf{s}(\boldsymbol{\theta}) \triangleq \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}) \quad (3.39)$$

The **Fisher information matrix (FIM)** is defined to be the covariance of the score function:

$$\mathbf{F}(\boldsymbol{\theta}) \triangleq \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x}|\boldsymbol{\theta})} [\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})^\top] \quad (3.40)$$

so the (i, j) ’th entry has the form

$$F_{ij} = \mathbb{E}_{\mathbf{x} \sim \boldsymbol{\theta}} \left[\left(\frac{\partial}{\partial \theta_i} \log p(\mathbf{x}|\boldsymbol{\theta}) \right) \left(\frac{\partial}{\partial \theta_j} \log p(\mathbf{x}|\boldsymbol{\theta}) \right) \right] \quad (3.41)$$

We give an interpretation of this quantity below.

3.3.4.2 Equivalence between the FIM and the Hessian of the NLL

In this section, we prove that the Fisher information matrix equals the expected Hessian of the negative log likelihood (NLL)

$$\text{NLL}(\boldsymbol{\theta}) = -\log p(\mathcal{D}|\boldsymbol{\theta}) \quad (3.42)$$

Since the Hessian measures the curvature of the likelihood, we see that the FIM tells us how well the likelihood function can identify the best set of parameters. (If a likelihood function is flat, we cannot infer anything about the parameters, but if it is a delta function at a single point, the best parameter vector will be uniquely determined.) Thus the FIM is intimately related to the frequentist notion of uncertainty of the MLE, which is captured by the variance we expect to see in the MLE if we were to compute it on multiple different datasets drawn from our model.

More precisely, we have the following theorem.

Theorem 3.3.2. *If $\log p(\mathbf{x}|\boldsymbol{\theta})$ is twice differentiable, and under certain regularity conditions, the FIM is equal to the expected Hessian of the NLL, i.e.,*

$$\mathbf{F}(\boldsymbol{\theta})_{ij} \triangleq \mathbb{E}_{\mathbf{x} \sim \boldsymbol{\theta}} \left[\left(\frac{\partial}{\partial \theta_i} \log p(\mathbf{x}|\boldsymbol{\theta}) \right) \left(\frac{\partial}{\partial \theta_j} \log p(\mathbf{x}|\boldsymbol{\theta}) \right) \right] = \mathbb{E}_{\mathbf{x} \sim \boldsymbol{\theta}} \left[\frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\mathbf{x}|\boldsymbol{\theta}) \right] \quad (3.43)$$

Before we prove this result, we establish the following important lemma.

Lemma 3.3.1. *The expected value of the score function is zero, i.e.,*

$$\mathbb{E}_{p(\mathbf{x}|\boldsymbol{\theta})} [\nabla \log p(\mathbf{x}|\boldsymbol{\theta})] = \mathbf{0} \quad (3.44)$$

We prove this lemma in the scalar case. First, note that since $\int p(x|\theta)dx = 1$, we have

$$\frac{\partial}{\partial \theta} \int p(x|\theta)dx = 0 \quad (3.45)$$

Combining this with the identity

$$\frac{\partial}{\partial \theta} p(x|\theta) = \left[\frac{\partial}{\partial \theta} \log p(x|\theta) \right] p(x|\theta) \quad (3.46)$$

we have

$$0 = \int \frac{\partial}{\partial \theta} p(x|\theta)dx = \int \left[\frac{\partial}{\partial \theta} \log p(x|\theta) \right] p(x|\theta)dx = \mathbb{E}[s(\theta)] \quad (3.47)$$

Now we return to the proof of our main theorem. For simplicity, we will focus on the scalar case, following the presentation of [Ric95, p263].

Proof. Taking derivatives of Equation (3.47), we have

$$0 = \frac{\partial}{\partial \theta} \int \left[\frac{\partial}{\partial \theta} \log p(x|\theta) \right] p(x|\theta)dx \quad (3.48)$$

$$= \int \left[\frac{\partial^2}{\partial \theta^2} \log p(x|\theta) \right] p(x|\theta)dx + \int \left[\frac{\partial}{\partial \theta} \log p(x|\theta) \right] \frac{\partial}{\partial \theta} p(x|\theta)dx \quad (3.49)$$

$$= \int \left[\frac{\partial^2}{\partial \theta^2} \log p(x|\theta) \right] p(x|\theta)dx + \int \left[\frac{\partial}{\partial \theta} \log p(x|\theta) \right]^2 p(x|\theta)dx \quad (3.50)$$

and hence

$$-\mathbb{E}_{x \sim \theta} \left[\frac{\partial^2}{\partial \theta^2} \log p(x|\theta) \right] = \mathbb{E}_{x \sim \theta} \left[\left(\frac{\partial}{\partial \theta} \log p(x|\theta) \right)^2 \right] \quad (3.51)$$

as claimed. \square

Now consider the Hessian of the NLL given N iid samples $\mathcal{D} = \{\mathbf{x}_n : n = 1 : N\}$:

$$H_{ij} \triangleq -\frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\mathcal{D}|\boldsymbol{\theta}) = -\sum_{n=1}^N \frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\mathbf{x}_n|\boldsymbol{\theta}) \quad (3.52)$$

From the above theorem, we have

$$\mathbb{E}_{p(\mathcal{D}|\boldsymbol{\theta})} [\mathbf{H}(\mathcal{D})|_{\boldsymbol{\theta}}] = N\mathbf{F}(\boldsymbol{\theta}) \quad (3.53)$$

This is useful when deriving the sampling distribution of the MLE, as discussed in Section 3.3.3.

3.3.4.3 Example: FIM for the binomial

Suppose $x \sim \text{Bin}(n, \theta)$. The log likelihood for a single sample is

$$l(\theta|x) = x \log \theta + (n-x) \log(1-\theta) \quad (3.54)$$

The score function is just the gradient of the log-likelihood:

$$s(\theta|x) \triangleq \frac{d}{d\theta} l(\theta|x) = \frac{x}{\theta} - \frac{n-x}{1-\theta} \quad (3.55)$$

The gradient of the score function is

$$s'(\theta|x) = -\frac{x}{\theta^2} - \frac{n-x}{(1-\theta)^2} \quad (3.56)$$

Hence the Fisher information is given by

$$F(\theta) = \mathbb{E}_{x \sim \theta} [-s'(\theta|x)] = \frac{n\theta}{\theta^2} + \frac{n-n\theta}{(1-\theta)^2} = \frac{n}{\theta} + \frac{n}{1-\theta} = \frac{n}{\theta(1-\theta)} \quad (3.57)$$

3.3.4.4 Example: FIM for the univariate Gaussian

Consider a univariate Gaussian $p(x|\boldsymbol{\theta}) = \mathcal{N}(x|\mu, v)$. We have

$$\ell(\boldsymbol{\theta}) = \log p(x|\boldsymbol{\theta}) = -\frac{1}{2v}(x-\mu)^2 - \frac{1}{2} \log(v) - \frac{1}{2} \log(2\pi) \quad (3.58)$$

The partial derivatives are given by

$$\frac{\partial \ell}{\partial \mu} = (x-\mu)v^{-1}, \quad \frac{\partial^2 \ell}{\partial \mu^2} = -v^{-1} \quad (3.59)$$

$$\frac{\partial \ell}{\partial v} = \frac{1}{2}v^{-2}(x-\mu)^2 - \frac{1}{2}v^{-1}, \quad \frac{\partial^2 \ell}{\partial v^2} = -v^{-3}(x-\mu)^2 + \frac{1}{2}v^{-2} \quad (3.60)$$

$$\frac{\partial \ell}{\partial \mu \partial v} = -v^{-2}(x-\mu) \quad (3.61)$$

and hence

$$\mathbf{F}(\boldsymbol{\theta}) = \begin{pmatrix} \mathbb{E}[v^{-1}] & \mathbb{E}[v^{-2}(x-\mu)] \\ \mathbb{E}[v^{-2}(x-\mu)] & \mathbb{E}[v^{-3}(x-\mu)^2 - \frac{1}{2}v^{-2}] \end{pmatrix} = \begin{pmatrix} \frac{1}{v} & 0 \\ 0 & \frac{1}{2v^2} \end{pmatrix} \quad (3.62)$$

3.3.4.5 Example: FIM for logistic regression

Consider ℓ_2 -regularized binary logistic regression. The negative log joint has the following form:

$$\mathcal{L}(\mathbf{w}) = -\log[p(\mathbf{y}|\mathbf{X}, \mathbf{w})p(\mathbf{w}|\lambda)] = -\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \sum_{n=1}^N \log(1 + e^{\mathbf{w}^\top \mathbf{x}_n}) + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w} \quad (3.63)$$

The derivative has the form

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = -\mathbf{X}^\top \mathbf{y} + \mathbf{X}^\top \mathbf{s} + \lambda \mathbf{w} \quad (3.64)$$

where $s_n = \sigma(\mathbf{w}^\top \mathbf{x}_n)$. The FIM is given by

$$\mathbf{F}(\mathbf{w}) = \mathbb{E}_{p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \lambda)} [\nabla^2 \mathcal{L}(\mathbf{w})] = \mathbf{X}^\top \mathbf{\Lambda} \mathbf{X} + \lambda \mathbf{I} \quad (3.65)$$

where $\mathbf{\Lambda}$ is the $N \times N$ diagonal matrix with entries

$$\Lambda_{nn} = \sigma(\mathbf{w}^\top \mathbf{x}_n)(1 - \sigma(\mathbf{w}^\top \mathbf{x}_n)) \quad (3.66)$$

3.3.4.6 FIM for the exponential family

In this section, we discuss how to derive the FIM for an exponential family distribution with natural parameters $\boldsymbol{\eta}$, which generalizes many of the previous examples. Recall from Equation (2.216) that the gradient of the log partition function is the expected sufficient statistics

$$\nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta}) = \mathbb{E}[\mathcal{T}(\mathbf{x})] = \mathbf{m} \quad (3.67)$$

and from Equation (2.247) that the gradient of the log likelihood is the statistics minus their expected value:

$$\nabla_{\boldsymbol{\eta}} \log p(\mathbf{x}|\boldsymbol{\eta}) = \mathcal{T}(\mathbf{x}) - \mathbb{E}[\mathcal{T}(\mathbf{x})] \quad (3.68)$$

Hence the FIM wrt the natural parameters $\mathbf{F}_{\boldsymbol{\eta}}$ is given by

$$(\mathbf{F}_{\boldsymbol{\eta}})_{ij} = \mathbb{E}_{p(\mathbf{x}|\boldsymbol{\eta})} \left[\frac{\partial \log p(\mathbf{x}|\boldsymbol{\eta})}{\partial \eta_i} \frac{\partial \log p(\mathbf{x}|\boldsymbol{\eta})}{\partial \eta_j} \right] \quad (3.69)$$

$$= \mathbb{E}_{p(\mathbf{x}|\boldsymbol{\eta})} [(\mathcal{T}(\mathbf{x})_i - m_i)(\mathcal{T}(\mathbf{x})_j - m_j)] \quad (3.70)$$

$$= \text{Cov}[\mathcal{T}(\mathbf{x})_i, \mathcal{T}(\mathbf{x})_j] \quad (3.71)$$

or, in short,

$$\mathbf{F}_{\boldsymbol{\eta}} = \text{Cov}[\mathcal{T}(\mathbf{x})] \quad (3.72)$$

Sometimes we need to compute the Fisher wrt the moment parameters \mathbf{m} :

$$(\mathbf{F}_{\mathbf{m}})_{ij} = \mathbb{E}_{p(\mathbf{x}|\mathbf{m})} \left[\frac{\partial \log p(\mathbf{x}|\boldsymbol{\eta})}{\partial m_i} \frac{\partial \log p(\mathbf{x}|\boldsymbol{\eta})}{\partial m_j} \right] \quad (3.73)$$

From the chain rule we have

$$\frac{\partial \log p(\mathbf{x})}{\partial \alpha} = \frac{\partial \log p(\mathbf{x})}{\partial \beta} \frac{\partial \beta}{\partial \alpha} \quad (3.74)$$

and hence

$$\mathbf{F}_\alpha = \frac{\partial \boldsymbol{\beta}^\top}{\partial \boldsymbol{\alpha}} \mathbf{F}_{\boldsymbol{\beta}} \frac{\partial \boldsymbol{\beta}}{\partial \boldsymbol{\alpha}} \quad (3.75)$$

Using the log trick

$$\nabla \mathbb{E}_{p(\mathbf{x})} [f(\mathbf{x})] = \mathbb{E}_{p(\mathbf{x})} [f(\mathbf{x}) \nabla \log p(\mathbf{x})] \quad (3.76)$$

and Equation (3.68) we have

$$\frac{\partial m_i}{\partial \eta_j} = \frac{\partial \mathbb{E} [\mathcal{T}(\mathbf{x})_i]}{\partial \eta_j} = \mathbb{E} \left[\mathcal{T}(\mathbf{x})_i \frac{\partial \log p(\mathbf{x}|\boldsymbol{\eta})}{\partial \eta_j} \right] = \mathbb{E} [\mathcal{T}(\mathbf{x})_i (\mathcal{T}(\mathbf{x})_j - m_j)] \quad (3.77)$$

$$= \mathbb{E} [\mathcal{T}(\mathbf{x})_i \mathcal{T}(\mathbf{x})_j] - \mathbb{E} [\mathcal{T}(\mathbf{x})_i] m_j = \text{Cov} [\mathcal{T}(\mathbf{x})_i \mathcal{T}(\mathbf{x})_j] = (\mathbf{F}_{\boldsymbol{\eta}})_{ij} \quad (3.78)$$

and hence

$$\frac{\partial \boldsymbol{\eta}}{\partial \mathbf{m}} = \mathbf{F}_{\boldsymbol{\eta}}^{-1} \quad (3.79)$$

so

$$\mathbf{F}_{\mathbf{m}} = \frac{\partial \boldsymbol{\eta}^\top}{\partial \mathbf{m}} \mathbf{F}_{\boldsymbol{\eta}} \frac{\partial \boldsymbol{\eta}}{\partial \mathbf{m}} = \mathbf{F}_{\boldsymbol{\eta}}^{-1} \mathbf{F}_{\boldsymbol{\eta}} \mathbf{F}_{\boldsymbol{\eta}}^{-1} = \mathbf{F}_{\boldsymbol{\eta}}^{-1} = \text{Cov} [\mathcal{T}(\mathbf{x})]^{-1} \quad (3.80)$$

3.3.5 Counterintuitive properties of frequentist statistics

Although the frequentist approach to statistics is widely taught, it suffers from certain pathological properties, resulting in its often being misunderstood and/or misused, as has been pointed out in multiple articles (see e.g., [Bol02; Bri12; Cla21; Gel16; Hoe+14; Jay03; Kru10; Lav00; Lyu+20; Min99; Mac03; WG17]). We give some examples below.

3.3.5.1 Confidence intervals

In frequentist statistics, we use the variability induced by the sampling distribution as a way to estimate uncertainty of a parameter estimate. In particular, we define a $100(1 - \alpha)\%$ **confidence interval** as any interval $I(\tilde{\mathcal{D}}) = (\ell(\tilde{\mathcal{D}}), u(\tilde{\mathcal{D}}))$ derived from a hypothetical dataset $\tilde{\mathcal{D}}$ such that

$$\Pr(\theta \in I(\tilde{\mathcal{D}}) | \tilde{\mathcal{D}} \sim \theta) = 1 - \alpha \quad (3.81)$$

It is common to set $\alpha = 0.05$, which yields a 95% CI. This means that, if we repeatedly sampled data, and compute $I(\tilde{\mathcal{D}})$ for each such dataset, then about 95% of such intervals will contain the true parameter θ . We say that the CI has 95% **coverage**.

Note, however, that Equation (3.81) does *not* mean that for any particular dataset that $\theta \in I(\mathcal{D})$ with 95% probability, which is what a Bayesian credible interval computes (Section 3.2.1.7), and

which is what most people are usually interested in. So we see that the concept of frequentist CI and Bayesian CI are quite different: In the frequentist approach, θ is treated as an unknown fixed constant, and the data is treated as random. In the Bayesian approach, we treat the data as fixed (since it is known) and the parameter as random (since it is unknown).

This counter-intuitive definition of confidence intervals can lead to bizarre results. Consider the following example from [Ber85a, p11]. Suppose we draw two integers $\mathcal{D} = (y_1, y_2)$ from

$$p(y|\theta) = \begin{cases} 0.5 & \text{if } y = \theta \\ 0.5 & \text{if } y = \theta + 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.82)$$

If $\theta = 39$, we would expect the following outcomes each with probability 0.25:

$$(39, 39), (39, 40), (40, 39), (40, 40) \quad (3.83)$$

Let $m = \min(y_1, y_2)$ and define the following interval:

$$[\ell(\mathcal{D}), u(\mathcal{D})] = [m, m] \quad (3.84)$$

For the above samples this yields

$$[39, 39], [39, 39], [39, 39], [40, 40] \quad (3.85)$$

Hence Equation (3.84) is clearly a 75% CI, since 39 is contained in 3/4 of these intervals. However, if we observe $\mathcal{D} = (39, 40)$ then $p(\theta = 39|\mathcal{D}) = 1.0$, so we know that θ must be 39, yet we only have 75% “confidence” in this fact. We see that the CI will “cover” the true parameter 75% of the time, if we compute multiple CIs from different randomly sampled datasets, but if we just have a single observed dataset, and hence a single CI, then the frequentist “coverage” probability can be very misleading.

Several more interesting examples, along with Python code, can be found at [Van14]. See also [Hoe+14; Mor+16; Lyu+20; Cha+19b], who show that many people, including professional statisticians, misunderstand and misuse frequentist confidence intervals in practice, whereas Bayesian credible intervals do not suffer from these problems.

3.3.5.2 p-values

The frequentist approach to hypothesis testing, known as **null hypothesis significance testing** or **NHST**, is to define a decision procedure for deciding whether to accept or reject the **null hypothesis** H_0 based on whether some observed **test statistic** $t(\mathcal{D})$ is likely or not under the sampling distribution of the null model. We describe this procedure in more detail in Section 3.10.1.

Rather than accepting or rejecting the null hypothesis, we can compute a quantity related to how likely the null hypothesis is to be true. In particular, we can compute a quantity called a **p-value**, which is defined as

$$\text{pval}(t(\mathcal{D})) \triangleq \Pr(t(\tilde{\mathcal{D}}) \geq t(\mathcal{D}) | \tilde{\mathcal{D}} \sim H_0) \quad (3.86)$$

where $\tilde{\mathcal{D}} \sim H_0$ is hypothetical future data. That is, the p-value is just the tail probability of observing the value $t(\mathcal{D})$ under the sampling distribution. (Note that the p-value does not explicitly depend

on a model of the data, but most common test statistics implicitly define a model, as we discuss in Section 3.10.3.)

A p-value is often interpreted as the likelihood of the data under the null hypothesis, so small values are interpreted to mean that H_0 is unlikely, and therefore that H_1 is likely. The reasoning is roughly as follows:

If H_0 is true, then this test statistic would probably not occur. This statistic did occur. Therefore H_0 is probably false.

However, this is invalid reasoning. To see why, consider the following example (from [Coh94]):

If a person is an American, then he is probably not a member of Congress. This person is a member of Congress. Therefore he is probably not an American.

This is obviously fallacious reasoning. By contrast, the following logical argument is valid reasoning:

If a person is a Martian, then he is not a member of Congress. This person is a member of Congress. Therefore he is not a Martian.

The difference between these two cases is that the Martian example is using **deduction**, that is, reasoning forward from logical definitions to their consequences. More precisely, this example uses a rule from logic called **modus tollens**, in which we start out with a definition of the form $P \Rightarrow Q$; when we observe $\neg Q$, we can conclude $\neg P$. By contrast, the American example concerns **induction**, that is, reasoning backwards from observed evidence to probable (but not necessarily true) causes using statistical regularities, not logical definitions.

To perform induction, we need to use probabilistic inference (as explained in detail in [Jay03]). In particular, to compute the probability of the null hypothesis, we should use Bayes rule, as follows:

$$p(H_0|\mathcal{D}) = \frac{p(\mathcal{D}|H_0)p(H_0)}{p(\mathcal{D}|H_0)p(H_0) + p(\mathcal{D}|H_1)p(H_1)} \quad (3.87)$$

If the prior is uniform, so $p(H_0) = p(H_1) = 0.5$, this can be rewritten in terms of the **likelihood ratio** $LR = p(\mathcal{D}|H_0)/p(\mathcal{D}|H_1)$ as follows:

$$p(H_0|\mathcal{D}) = \frac{LR}{LR + 1} \quad (3.88)$$

In the American Congress example, \mathcal{D} is the observation that the person is a member of Congress. The null hypothesis H_0 is that the person is American, and the alternative hypothesis H_1 is that the person is not American. We assume that $p(\mathcal{D}|H_0)$ is low, since most Americans are not members of Congress. However, $p(\mathcal{D}|H_1)$ is also low — in fact, in this example, it is 0, since only Americans can be members of Congress. Hence $LR = \infty$, so $p(H_0|\mathcal{D}) = 1.0$, as intuition suggests.

Note, however, that NHST ignores $p(\mathcal{D}|H_1)$ as well as the prior $p(H_0)$, so it gives the wrong results, not just in this problem, but in many problems. Indeed, even most scientists misinterpret p-values.². Consequently the journal *The American Statistician* published a whole special issue warning about the use of p-values and NHST [WSL19], and several journals have even banned p-values [TM15; AGM19].

2. See e.g., <https://fivethirtyeight.com/features/not-even-scientists-can-easily-explain-p-values/>.

3.3.5.3 Discussion

The above problems stem from the fact that frequentist inference is not conditional on the actually observed data, but instead is based on properties derived from the sampling distribution of the estimator. However, conditional probability statements are what most people want. As Jim Berger writes in [Ber85a]:

Users of statistics want to know the probability (after seeing the data) that a hypothesis is true, or the probability that θ is in a given interval, and yet classical statistics does not allow one to talk of such things. Instead, artificial concepts such as error probabilities and coverage probabilities are introduced as substitutes. It is ironic that non-Bayesians often claim that the Bayesians form a dogmatic unrealistic religion, when instead it is the non-Bayesian methods that are often founded on elaborate and artificial structures. Unfortunately, those who become used to these artificial structures come to view them as natural, and hence this line of argument tends to have little effect on the established non-Bayesian. — Jim Berger, [Ber85a].

3.3.6 Why isn't everyone a Bayesian?

I believe that it would be very difficult to persuade an intelligent person that current [frequentist] statistical practice was sensible, but that there would be much less difficulty with an approach via likelihood and Bayes' theorem. — George Box, 1962 (quoted in [Jay76]).

In Section 3.3.5 we showed that inference based on frequentist principles can exhibit various forms of counterintuitive behavior that can sometimes contradict common sense. Given these problems of frequentist statistics, an obvious question to ask is: “Why isn’t everyone a Bayesian?” The statistician Bradley Efron wrote a paper with exactly this title [Efr86]. His short paper is well worth reading for anyone interested in this topic. Below we quote his opening section:

The title is a reasonable question to ask on at least two counts. First of all, everyone used to be a Bayesian. Laplace wholeheartedly endorsed Bayes's formulation of the inference problem, and most 19th-century scientists followed suit. This included Gauss, whose statistical work is usually presented in frequentist terms.

A second and more important point is the cogency of the Bayesian argument. Modern statisticians, following the lead of Savage and de Finetti, have advanced powerful theoretical arguments for preferring Bayesian inference. A byproduct of this work is a disturbing catalogue of inconsistencies in the frequentist point of view.

Nevertheless, everyone is not a Bayesian. The current era (1986) is the first century in which statistics has been widely used for scientific reporting, and in fact, 20th-century statistics is mainly non-Bayesian. However, Lindley (1975) predicts a change for the 21st century.

Time will tell whether Lindley was right. However, the trends seem to be going in this direction. Traditionally, computation has been a barrier to using Bayesian methods, but this is less of an issue these days, due to faster computers and better algorithms, which we discuss in Part II.

Another, more fundamental, concern is that the Bayesian approach is only as correct as its modeling assumptions. In particular, it is important to check sensitivity of the conclusions to the choice of prior (and likelihood), using techniques such as Bayesian model checking (Section 3.9.1). In particular, as

Donald Rubin wrote in his paper called “Bayesianly Justifiable and Relevant Frequency Calculations for the Applied Statistician” [Rub84]:

The applied statistician should be Bayesian in principle and calibrated to the real world in practice. [They] should attempt to use specifications that lead to approximately calibrated procedures under reasonable deviations from [their assumptions]. [They] should avoid models that are contradicted by observed data in relevant ways — frequency calculations for hypothetical replications can model a model’s adequacy and help to suggest more appropriate models.

A final issue is more practical. Most users of statistical methods are not experts in statistics, but instead are experts in their own domain, such as psychology or social science. They often just want a simple (and fast!) method for testing a hypothesis, and so they turn to standard “cookie cutter” frequentist procedures, such as t -tests and χ^2 -tests. Fortunately there are simple Bayesian alternatives to these tests, as we discuss in Section 3.10, which avoid the conceptual problems we discussed in Section 3.3.5, and which can also be easily “upgraded” to use more complex (and realistic) modeling assumptions when necessary. Furthermore, by using an empirical Bayes approach, it is possible to derive automatic and robust Bayesian methods that have good frequentist properties but which are also conditional on the data, thus providing the best of both worlds.

For a more detailed discussion of the pros and cons of the Bayesian approach, specifically in the context of machine learning, see <https://bit.ly/3Rbd4lo> and <https://bit.ly/3j8miSR>.

3.4 Conjugate priors

In this section, we consider Bayesian inference for a class of models with a special form of prior, known as a **conjugate prior**, which simplifies the computation of the posterior. Formally, we say that a prior $p(\boldsymbol{\theta}) \in \mathcal{F}$ is a conjugate prior for a likelihood function $p(\mathcal{D}|\boldsymbol{\theta})$ if the posterior is in the same parameterized family as the prior, i.e., $p(\boldsymbol{\theta}|\mathcal{D}) \in \mathcal{F}$. In other words, \mathcal{F} is closed under Bayesian updating. If the family \mathcal{F} corresponds to the exponential family (defined in Section 2.4), then the computations can be performed in closed form. In more complex settings, we cannot perform closed-form inference, but we can often leverage these results as tractable subroutines inside of a larger computational pipeline.

3.4.1 The binomial model

One of the simplest examples of conjugate Bayesian analysis is the beta-binomial model. This is covered in detail in Section 3.2.1.

3.4.2 The multinomial model

In this section, we generalize the results from Section 3.4.1 from binary variables (e.g., coins) to K -ary variables (e.g., dice). Let $y \sim \text{Cat}(\boldsymbol{\theta})$ be a discrete random variable drawn from a **categorical distribution**. The likelihood has the form

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^N \text{Cat}(y_n|\boldsymbol{\theta}) = \prod_{n=1}^N \prod_{c=1}^C \theta_c^{\mathbb{I}(y_n=c)} = \prod_{c=1}^C \theta_c^{N_c} \quad (3.89)$$

where $N_c = \sum_n \mathbb{I}(y_n = c)$. We can generalize this to the **multinomial distribution** by defining $\mathbf{y} \sim \mathcal{M}(N, \boldsymbol{\theta})$, where N is the number of trials, and $y_c = N_c$ is the number of times value c is observed. The likelihood becomes

$$p(\mathbf{y}|N, \boldsymbol{\theta}) = \binom{N}{N_1 \dots N_C} \prod_{c=1}^C \theta_c^{N_c} \quad (3.90)$$

This is the same as the categorical likelihood modulo a scaling factor. Going forwards, we will work with the categorical model, for notational simplicity.

The conjugate prior for a categorical distribution is the Dirichlet distribution, which we discussed in Section 2.2.5.7. We denote this by $p(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\theta} | \boldsymbol{\alpha})$, where $\boldsymbol{\alpha}$ is the vector of prior pseudo-counts. Often we use a symmetric Dirichlet prior of the form $\alpha_k = \bar{\alpha}/K$. In this case, we have $\mathbb{E}[\theta_k] = 1/K$, and $\text{V}[\theta_k] = \frac{K-1}{K^2(\bar{\alpha}+1)}$. Thus we see that increasing the prior sample size $\bar{\alpha}$ decreases the variance of the prior, which is equivalent to using a stronger prior.

We can combine the multinomial likelihood and Dirichlet prior to compute the Dirichlet posterior, as follows:

$$p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})\text{Dir}(\boldsymbol{\theta} | \boldsymbol{\alpha}) \propto \left[\prod_k \theta_k^{N_k} \right] \left[\prod_k \theta_k^{\bar{\alpha}_k - 1} \right] \quad (3.91)$$

$$\propto \text{Dir}(\boldsymbol{\theta} | \bar{\alpha}_1 + N_1, \dots, \bar{\alpha}_K + N_K) = \text{Dir}(\boldsymbol{\theta} | \hat{\boldsymbol{\alpha}}) \quad (3.92)$$

where $\hat{\alpha}_k = \bar{\alpha}_k + N_k$ are the parameters of the posterior. So we see that the posterior can be computed by adding the empirical counts to the prior counts. In particular, the posterior mode is given by

$$\hat{\theta}_k = \frac{\hat{\alpha}_k - 1}{\sum_{k'=1}^K \hat{\alpha}_k - 1} = \frac{N_k + \bar{\alpha}_k - 1}{\sum_{k'=1}^K N_k + \bar{\alpha}_k - 1} \quad (3.93)$$

If we set $\alpha_k = 1$ we recover the MLE; if we set $\alpha_k = 2$, we recover the add-one smoothing estimate.

The marginal likelihood for the Dirichlet-categorical model is given by the following:

$$p(\mathcal{D}) = \frac{B(\mathbf{N} + \boldsymbol{\alpha})}{B(\boldsymbol{\alpha})} \quad (3.94)$$

where

$$B(\boldsymbol{\alpha}) = \frac{\prod_{k=1}^K \Gamma(\alpha_k)}{\Gamma(\sum_k \alpha_k)} \quad (3.95)$$

Hence we can rewrite the above result in the following form, which is what is usually presented in the literature:

$$p(\mathcal{D}) = \frac{\Gamma(\sum_k \alpha_k)}{\Gamma(N + \sum_k \alpha_k)} \prod_k \frac{\Gamma(N_k + \alpha_k)}{\Gamma(\alpha_k)} \quad (3.96)$$

For more details on this model, see [Mur22, Sec 4.6.3].

3.4.3 The univariate Gaussian model

In this section, we derive the posterior $p(\mu, \sigma^2 | \mathcal{D})$ for a univariate Gaussian. For simplicity, we consider this in three steps: inferring just μ , inferring just σ^2 , and then inferring both. See Section 3.4.4 for the multivariate case.

3.4.3.1 Posterior of μ given σ^2

If σ^2 is a known constant, the likelihood for μ has the form

$$p(\mathcal{D}|\mu) \propto \exp\left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \mu)^2\right) \quad (3.97)$$

One can show that the conjugate prior is another Gaussian, $\mathcal{N}(\mu | \check{m}, \check{\tau}^2)$. Applying Bayes' rule for Gaussians (Equation (2.121)), we find that the corresponding posterior is given by

$$p(\mu | \mathcal{D}, \sigma^2) = \mathcal{N}(\mu | \hat{m}, \hat{\tau}^2) \quad (3.98)$$

$$\hat{\tau}^2 = \frac{1}{\frac{N}{\sigma^2} + \frac{1}{\check{\tau}^2}} = \frac{\sigma^2 \check{\tau}^2}{N \check{\tau}^2 + \sigma^2} \quad (3.99)$$

$$\hat{m} = \check{\tau}^2 \left(\frac{\check{m}}{\check{\tau}^2} + \frac{N\bar{y}}{\sigma^2} \right) = \frac{\sigma^2}{N \check{\tau}^2 + \sigma^2} \check{m} + \frac{N \check{\tau}^2}{N \check{\tau}^2 + \sigma^2} \bar{y} \quad (3.100)$$

where $\bar{y} \triangleq \frac{1}{N} \sum_{n=1}^N y_n$ is the empirical mean.

This result is easier to understand if we work in terms of the precision parameters, which are just inverse variances. Specifically, let $\lambda = 1/\sigma^2$ be the observation precision, and $\check{\lambda} = 1/\check{\tau}^2$ be the precision of the prior. We can then rewrite the posterior as follows:

$$p(\mu | \mathcal{D}, \lambda) = \mathcal{N}(\mu | \hat{m}, \hat{\lambda}^{-1}) \quad (3.101)$$

$$\hat{\lambda} = \check{\lambda} + N\lambda \quad (3.102)$$

$$\hat{m} = \frac{N\lambda\bar{y} + \check{\lambda}\check{m}}{\hat{\lambda}} = \frac{N\lambda}{N\lambda + \check{\lambda}} \bar{y} + \frac{\check{\lambda}}{N\lambda + \check{\lambda}} \check{m} \quad (3.103)$$

These equations are quite intuitive: the posterior precision $\hat{\lambda}$ is the prior precision $\check{\lambda}$ plus N units of measurement precision λ . Also, the posterior mean \hat{m} is a convex combination of the empirical mean \bar{y} and the prior mean \check{m} . This makes it clear that the posterior mean is a compromise between the empirical mean and the prior. If the prior is weak relative to the signal strength ($\check{\lambda}$ is small relative to λ), we put more weight on the empirical mean. If the prior is strong relative to the signal strength ($\check{\lambda}$ is large relative to λ), we put more weight on the prior. This is illustrated in Figure 3.5. Note also that the posterior mean is written in terms of $N\lambda\bar{x}$, so having N measurements each of precision λ is like having one measurement with value \bar{x} and precision $N\lambda$.

To gain further insight into these equations, consider the posterior after seeing a single datapoint

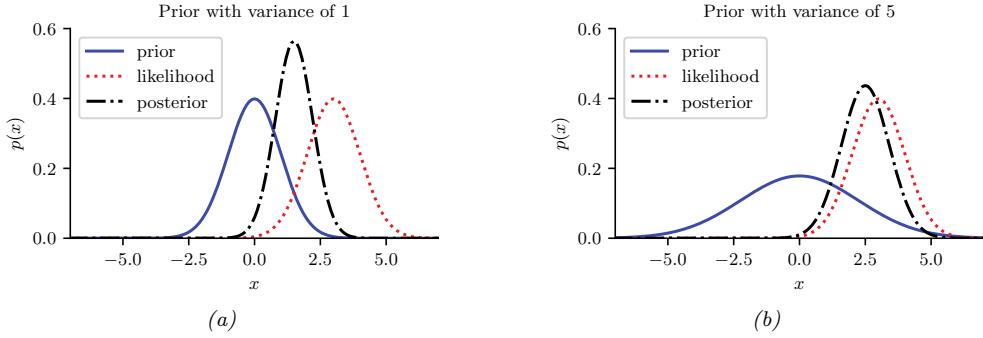


Figure 3.5: Inferring the mean of a univariate Gaussian with known σ^2 . (a) Using strong prior, $p(\mu) = \mathcal{N}(\mu|0, 1)$. (b) Using weak prior, $p(\mu) = \mathcal{N}(\mu|0, 5)$. Generated by [gauss_infer_1d.ipynb](#).

y (so $N = 1$). Then the posterior mean can be written in the following equivalent ways:

$$\hat{m} = \frac{\check{\lambda}}{\check{\lambda}} \check{m} + \frac{\lambda}{\check{\lambda}} y \quad (3.104)$$

$$= \check{m} + \frac{\lambda}{\check{\lambda}} (y - \check{m}) \quad (3.105)$$

$$= y - \frac{\lambda}{\check{\lambda}} (y - \check{m}) \quad (3.106)$$

The first equation is a convex combination of the prior mean and the data. The second equation is the prior mean adjusted towards the data y . The third equation is the data adjusted towards the prior mean; this is called a **shrinkage** estimate. This is easier to see if we define the weight $w = \check{\lambda}/\check{\lambda}$. Then we have

$$\hat{m} = y - w(y - \check{m}) = (1 - w)y + w\check{m} \quad (3.107)$$

Note that, for a Gaussian, the posterior mean and posterior mode are the same. Thus we can use the above equations to perform MAP estimation.

3.4.3.2 Posterior of σ^2 given μ

If μ is a known constant, the likelihood for σ^2 has the form

$$p(\mathcal{D}|\sigma^2) \propto (\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \mu)^2\right) \quad (3.108)$$

where we can no longer ignore the $1/(\sigma^2)$ term in front. The standard conjugate prior is the inverse gamma distribution (Section 2.2.3.4), given by

$$\text{IG}(\sigma^2 | \check{a}, \check{b}) = \frac{\check{b}^{\check{a}}}{\Gamma(\check{a})} (\sigma^2)^{-(\check{a}+1)} \exp(-\frac{\check{b}}{\sigma^2}) \quad (3.109)$$

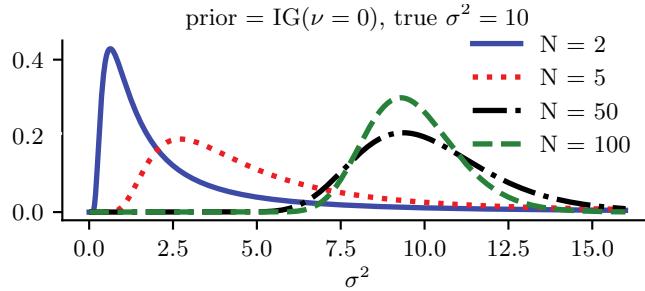


Figure 3.6: Sequential updating of the posterior for σ^2 starting from an uninformative prior. The data was generated from a Gaussian with known mean $\mu = 5$ and unknown variance $\sigma^2 = 10$. Generated by [gauss_seq_update_sigma_1d.ipynb](#)

Multiplying the likelihood and the prior, we see that the posterior is also IG:

$$p(\sigma^2 | \mu, \mathcal{D}) = \text{IG}(\sigma^2 | \hat{a}, \hat{b}) \quad (3.110)$$

$$\hat{a} = \check{a} + N/2 \quad (3.111)$$

$$\hat{b} = \check{b} + \frac{1}{2} \sum_{n=1}^N (y_n - \mu)^2 \quad (3.112)$$

See Figure 3.6 for an illustration.

One small annoyance with using the $\text{IG}(\check{a}, \check{b})$ distribution is that the strength of the prior is encoded in both \check{a} and \check{b} . Therefore, in the Bayesian statistics literature it is common to use an alternative parameterization of the IG distribution, known as the (scaled) **inverse chi-squared distribution**:

$$\chi^{-2}(\sigma^2 | \check{\nu}, \check{\tau}^2) = \text{IG}(\sigma^2 | \frac{\check{\nu}}{2}, \frac{\check{\nu} \check{\tau}^2}{2}) \propto (\sigma^2)^{-\check{\nu}/2-1} \exp(-\frac{\check{\nu} \check{\tau}^2}{2\sigma^2}) \quad (3.113)$$

Here $\check{\nu}$ (called the degrees of freedom or dof parameter) controls the strength of the prior, and $\check{\tau}^2$ encodes the prior mean. With this prior, the posterior becomes

$$p(\sigma^2 | \mathcal{D}, \mu) = \chi^{-2}(\sigma^2 | \hat{\nu}, \hat{\tau}^2) \quad (3.114)$$

$$\hat{\nu} = \check{\nu} + N \quad (3.115)$$

$$\hat{\tau}^2 = \frac{\check{\nu} \check{\tau}^2 + \sum_{n=1}^N (y_n - \mu)^2}{\hat{\nu}} \quad (3.116)$$

We see that the posterior dof $\hat{\nu}$ is the prior dof $\check{\nu}$ plus N , and the posterior sum of squares $\hat{\nu} \hat{\tau}^2$ is the prior sum of squares $\check{\nu} \check{\tau}^2$ plus the data sum of squares.

3.4.3.3 Posterior of μ and σ^2 : conjugate prior

Now suppose we want to infer both the mean and variance. The corresponding conjugate prior is the **normal inverse gamma**:

$$\text{NIG}(\mu, \sigma^2 | \check{m}, \check{\kappa}, \check{a}, \check{b}) \triangleq \mathcal{N}(\mu | \check{m}, \sigma^2 / \check{\kappa}) \text{IG}(\sigma^2 | \check{a}, \check{b}) \quad (3.117)$$

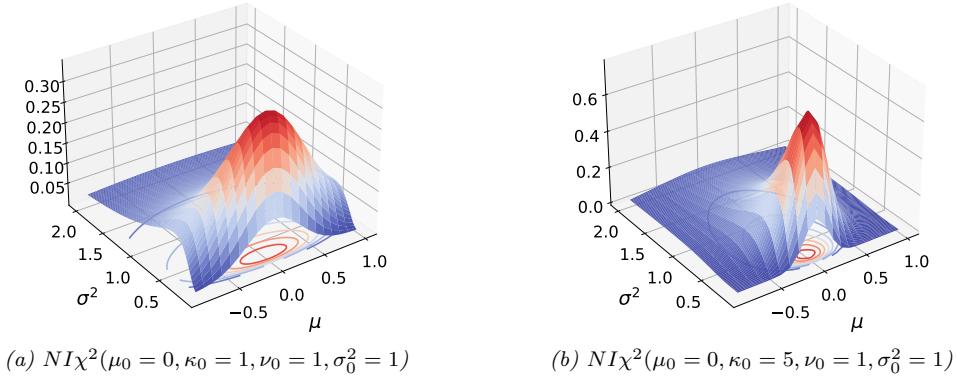


Figure 3.7: The $NI\chi^2(\mu, \sigma^2 | m, \kappa, \nu, \sigma^2)$ distribution. m is the prior mean and κ is how strongly we believe this; σ^2 is the prior variance and ν is how strongly we believe this. (a) $m = 0, \kappa = 1, \nu = 1, \sigma^2 = 1$. Notice that the contour plot (underneath the surface) is shaped like a “squashed egg”. (b) We increase the strength of our belief in the mean by setting $\kappa = 5$, so the distribution for μ around $m = 0$ becomes narrower. Generated by [nix_plots.ipynb](#).

However, it is common to use a reparameterization of this known as the **normal inverse chi-squared** or **NIX** distribution [Gel+14a, p67], which is defined by

$$NI\chi^2(\mu, \sigma^2 | \tilde{m}, \tilde{\kappa}, \tilde{\nu}, \tilde{\tau}^2) \triangleq \mathcal{N}(\mu | \tilde{m}, \sigma^2 / \tilde{\kappa}) \chi^{-2}(\sigma^2 | \tilde{\nu}, \tilde{\tau}^2) \quad (3.118)$$

$$\propto \left(\frac{1}{\sigma^2} \right)^{(\tilde{\nu}+3)/2} \exp \left(-\frac{\tilde{\nu}\tilde{\tau}^2 + \tilde{\kappa}(\mu - \tilde{m})^2}{2\sigma^2} \right) \quad (3.119)$$

See Figure 3.7 for some plots. Along the μ axis, the distribution is shaped like a Gaussian, and along the σ^2 axis, the distribution is shaped like a χ^{-2} ; the contours of the joint density have a “squashed egg” appearance. Interestingly, we see that the contours for μ are more peaked for small values of σ^2 , which makes sense, since if the data is low variance, we will be able to estimate its mean more reliably.

One can show (based on Section 3.4.4.3) that the posterior is given by

$$p(\mu, \sigma^2 | \mathcal{D}) = NI\chi^2(\mu, \sigma^2 | \hat{m}, \hat{\kappa}, \hat{\nu}, \hat{\tau}^2) \quad (3.120)$$

$$\hat{m} = \frac{\tilde{\kappa}\tilde{m} + N\bar{x}}{\tilde{\kappa}} \quad (3.121)$$

$$\hat{\kappa} = \tilde{\kappa} + N \quad (3.122)$$

$$\hat{\nu} = \tilde{\nu} + N \quad (3.123)$$

$$\hat{\tau}^2 = \tilde{\nu}\tilde{\tau}^2 + \sum_{n=1}^N (y_n - \bar{y})^2 + \frac{N \tilde{\kappa}}{\tilde{\kappa} + N} (\tilde{m} - \bar{y})^2 \quad (3.124)$$

The interpretation of this is as follows. For μ , the posterior mean \hat{m} is a convex combination of the prior mean \tilde{m} and the MLE \bar{x} ; the strength of this posterior, $\hat{\kappa}$, is the prior strength $\tilde{\kappa}$ plus the

number of datapoints N . For σ^2 , we work instead with the sum of squares: the posterior sum of squares, $\hat{\nu}\hat{\tau}^2$, is the prior sum of squares $\check{\nu}\check{\tau}^2$ plus the data sum of squares, $\sum_{n=1}^N(y_n - \bar{y})^2$, plus a term due to the discrepancy between the prior mean \check{m} and the MLE \bar{y} . The strength of this posterior, $\hat{\nu}$, is the prior strength $\check{\nu}$ plus the number of datapoints N ;

The posterior marginal for σ^2 is just

$$p(\sigma^2|\mathcal{D}) = \int p(\mu, \sigma^2|\mathcal{D})d\mu = \chi^{-2}(\sigma^2|\hat{\nu}, \hat{\tau}^2) \quad (3.125)$$

with the posterior mean given by $\mathbb{E}[\sigma^2|\mathcal{D}] = \frac{\hat{\nu}}{\hat{\nu}-2} \hat{\tau}^2$.

The posterior marginal for μ has a Student distribution, which follows from the fact that the Student distribution is a (scaled) mixture of Gaussians:

$$p(\mu|\mathcal{D}) = \int p(\mu, \sigma^2|\mathcal{D})d\sigma^2 = \mathcal{T}(\mu|\hat{m}, \hat{\tau}^2 / \hat{\kappa}, \hat{\nu}) \quad (3.126)$$

with the posterior mean given by $\mathbb{E}[\mu|\mathcal{D}] = \hat{m}$.

3.4.3.4 Posterior of μ and σ^2 : uninformative prior

If we “know nothing” about the parameters a priori, we can use an uninformative prior. We discuss how to create such priors in Section 3.5. A common approach is to use a Jeffreys prior. In Section 3.5.2.3, we show that the Jeffreys prior for a location and scale parameter has the form

$$p(\mu, \sigma^2) \propto p(\mu)p(\sigma^2) \propto \sigma^{-2} \quad (3.127)$$

We can simulate this with a conjugate prior by using

$$p(\mu, \sigma^2) = NI\chi^2(\mu, \sigma^2 | \check{m}=0, \check{\kappa}=0, \check{\nu}=-1, \check{\tau}^2=0) \quad (3.128)$$

With this prior, the posterior has the form

$$p(\mu, \sigma^2|\mathcal{D}) = NI\chi^2(\mu, \sigma^2 | \hat{m}=\bar{y}, \hat{\kappa}=N, \hat{\nu}=N-1, \hat{\tau}^2=s^2) \quad (3.129)$$

where

$$s^2 \triangleq \frac{1}{N-1} \sum_{n=1}^N (y_n - \bar{y})^2 = \frac{N}{N-1} \hat{\sigma}_{\text{mle}}^2 \quad (3.130)$$

s is known as the **sample standard deviation**. Hence the marginal posterior for the mean is given by

$$p(\mu|\mathcal{D}) = \mathcal{T}(\mu|\bar{y}, \frac{s^2}{N}, N-1) = \mathcal{T}(\mu|\bar{y}, \frac{\sum_{n=1}^N (y_n - \bar{y})^2}{N(N-1)}, N-1) \quad (3.131)$$

Thus the posterior variance of μ is

$$\mathbb{V}[\mu|\mathcal{D}] = \frac{\hat{\nu}}{\hat{\nu}-2} \hat{\tau}^2 = \frac{N-1}{N-3} \frac{s^2}{N} \rightarrow \frac{s^2}{N} \quad (3.132)$$

The square root of this is called the **standard error of the mean**:

$$\text{se}(\mu) \triangleq \sqrt{\mathbb{V}[\mu|\mathcal{D}]} \approx \frac{s}{\sqrt{N}} \quad (3.133)$$

Thus we can approximate the 95% **credible interval** for μ using

$$I_{.95}(\mu|\mathcal{D}) = \bar{y} \pm 2 \frac{s}{\sqrt{N}} \quad (3.134)$$

3.4.4 The multivariate Gaussian model

In this section, we derive the posterior $p(\boldsymbol{\mu}, \boldsymbol{\Sigma}|\mathcal{D})$ for a multivariate Gaussian. For simplicity, we consider this in three steps: inferring just $\boldsymbol{\mu}$, inferring just $\boldsymbol{\Sigma}$, and then inferring both.

3.4.4.1 Posterior of $\boldsymbol{\mu}$ given $\boldsymbol{\Sigma}$

The likelihood has the form

$$p(\mathcal{D}|\boldsymbol{\mu}) = \mathcal{N}(\bar{\mathbf{y}}|\boldsymbol{\mu}, \frac{1}{N}\boldsymbol{\Sigma}) \quad (3.135)$$

For simplicity, we will use a conjugate prior, which in this case is a Gaussian. In particular, if $p(\boldsymbol{\mu}) = \mathcal{N}(\boldsymbol{\mu}|\check{\mathbf{m}}, \check{\mathbf{V}})$ then we can derive a Gaussian posterior for $\boldsymbol{\mu}$ based on the results in Section 2.3.2.2. We get

$$p(\boldsymbol{\mu}|\mathcal{D}, \boldsymbol{\Sigma}) = \mathcal{N}(\boldsymbol{\mu}|\hat{\mathbf{m}}, \hat{\mathbf{V}}) \quad (3.136)$$

$$\hat{\mathbf{V}}^{-1} = \check{\mathbf{V}}^{-1} + N\boldsymbol{\Sigma}^{-1} \quad (3.137)$$

$$\hat{\mathbf{m}} = \hat{\mathbf{V}}(\boldsymbol{\Sigma}^{-1}(N\bar{\mathbf{y}}) + \check{\mathbf{V}}^{-1}\check{\mathbf{m}}) \quad (3.138)$$

Figure 3.8 gives a 2d example of these results.

3.4.4.2 Posterior of $\boldsymbol{\Sigma}$ given $\boldsymbol{\mu}$

We now discuss how to compute $p(\boldsymbol{\Sigma}|\mathcal{D}, \boldsymbol{\mu})$.

Likelihood

We can rewrite the likelihood as follows:

$$p(\mathcal{D}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \propto |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp\left(-\frac{1}{2}\text{tr}(\mathbf{S}_\mu \boldsymbol{\Sigma}^{-1})\right) \quad (3.139)$$

where

$$\mathbf{S}_\mu \triangleq \sum_{n=1}^N (\mathbf{y}_n - \boldsymbol{\mu})(\mathbf{y}_n - \boldsymbol{\mu})^\top \quad (3.140)$$

is the scatter matrix around $\boldsymbol{\mu}$.

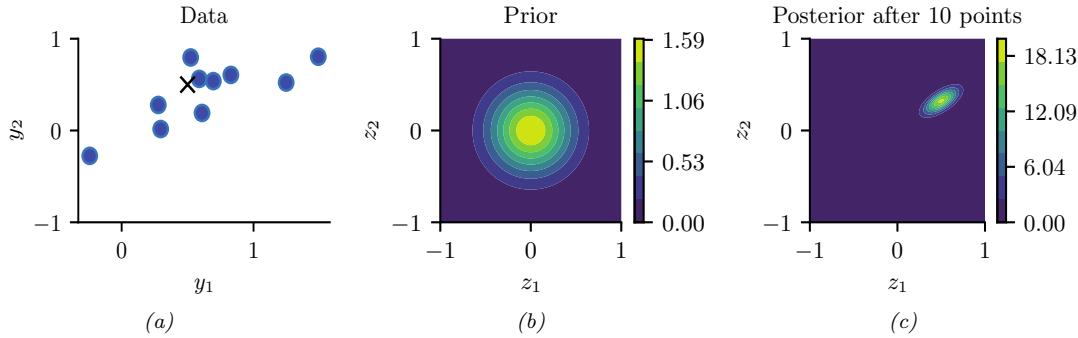


Figure 3.8: Illustration of Bayesian inference for a 2d Gaussian random vector \mathbf{z} . (a) The data is generated from $\mathbf{y}_n \sim \mathcal{N}(\mathbf{z}, \Sigma_y)$, where $\mathbf{z} = [0.5, 0.5]^\top$ and $\Sigma_y = 0.1([2, 1; 1, 1])$. We assume the sensor noise covariance Σ_y is known but \mathbf{z} is unknown. The black cross represents \mathbf{z} . (b) The prior is $p(\mathbf{z}) = \mathcal{N}(\mathbf{z} | \mathbf{0}, 0.1\mathbf{I}_2)$. (c) We show the posterior after 10 datapoints have been observed. Generated by `gauss_infer_2d.ipynb`.

Prior

The conjugate prior is known as the **inverse Wishart** distribution, which is a distribution over positive definite matrices, as we explained in Section 2.2.5.5. This has the following pdf:

$$\text{IW}(\boldsymbol{\Sigma} | \boldsymbol{\Psi}^{-1}, \boldsymbol{\nu}) \propto |\boldsymbol{\Sigma}|^{-(\nu + D + 1)/2} \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Psi} \boldsymbol{\Sigma}^{-1})\right) \quad (3.141)$$

Here $\check{\nu} > D - 1$ is the degrees of freedom (dof), and $\check{\Psi}$ is a symmetric pd matrix. We see that $\check{\Psi}$ plays the role of the prior scatter matrix, and $N_0 \triangleq \check{\nu} + D + 1$ controls the strength of the prior, and hence plays a role analogous to the sample size N .

Posterior

Multiplying the likelihood and prior we find that the posterior is also inverse Wishart:

$$p(\boldsymbol{\Sigma} | \mathcal{D}, \boldsymbol{\mu}) \propto |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Sigma}^{-1} \mathbf{S}_\mu)\right) |\boldsymbol{\Sigma}|^{-(\tilde{\nu} + D + 1)/2} \\ \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Sigma}^{-1} \tilde{\Psi})\right) \quad (3.142)$$

$$= |\Sigma|^{-\frac{N+(\Upsilon+D+1)}{2}} \exp \left(-\frac{1}{2} \text{tr} \left[\Sigma^{-1} (\mathbf{S}_\mu + \breve{\Psi}) \right] \right) \quad (3.143)$$

$$= \text{IW}(\Sigma | \hat{\Psi}, \hat{\nu}) \quad (3.144)$$

$$\hat{\nu} = \check{\nu} + N \quad (3.145)$$

$$\hat{\Psi} = \check{\Psi} + S_\mu \quad (3.146)$$

In words, this says that the posterior strength $\tilde{\nu}$ is the prior strength $\check{\nu}$ plus the number of observations N , and the posterior scatter matrix $\tilde{\Psi}$ is the prior scatter matrix $\check{\Psi}$ plus the data scatter matrix \mathbf{S}_μ .

3.4.4.3 Posterior of Σ and μ

In this section, we compute $p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D})$ using a conjugate prior.

Likelihood

The likelihood is given by

$$p(\mathcal{D} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) \propto |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp \left(-\frac{1}{2} \sum_{n=1}^N (\mathbf{y}_n - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{y}_n - \boldsymbol{\mu}) \right) \quad (3.147)$$

One can show that

$$\sum_{n=1}^N (\mathbf{y}_n - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{y}_n - \boldsymbol{\mu}) = \text{tr}(\boldsymbol{\Sigma}^{-1} \mathbf{S}) + N(\bar{\mathbf{y}} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\bar{\mathbf{y}} - \boldsymbol{\mu}) \quad (3.148)$$

where

$$\mathbf{S} \triangleq \mathbf{S}_{\bar{\mathbf{y}}} = \sum_{n=1}^N (\mathbf{y}_n - \bar{\mathbf{y}})(\mathbf{y}_n - \bar{\mathbf{y}})^\top = \mathbf{Y}^\top \mathbf{C}_N \mathbf{Y} \quad (3.149)$$

is empirical **scatter matrix**, and \mathbf{C}_N is the **centering matrix**

$$\mathbf{C}_N \triangleq \mathbf{I}_N - \frac{1}{N} \mathbf{1}_N \mathbf{1}_N^\top \quad (3.150)$$

Hence we can rewrite the likelihood as follows:

$$p(\mathcal{D} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) \propto |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp \left(-\frac{N}{2} (\boldsymbol{\mu} - \bar{\mathbf{y}})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - \bar{\mathbf{y}}) \right) \exp \left(-\frac{1}{2} \text{tr}(\mathbf{S} \boldsymbol{\Sigma}^{-1}) \right) \quad (3.151)$$

We will use this form below.

Prior

The obvious prior to use is the following

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\boldsymbol{\mu} | \check{\mathbf{m}}, \check{\mathbf{V}}) \text{IW}(\boldsymbol{\Sigma} | \check{\boldsymbol{\Psi}}^{-1}, \check{\nu}) \quad (3.152)$$

where IW is the inverse Wishart distribution. Unfortunately, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ appear together in a non-factorized way in the likelihood in Equation (3.151) (see the first exponent term), so the factored prior in Equation (3.152) is not conjugate to the likelihood.³

The above prior is sometimes called **conditionally conjugate**, since both conditionals, $p(\boldsymbol{\mu} | \boldsymbol{\Sigma})$ and $p(\boldsymbol{\Sigma} | \boldsymbol{\mu})$, are individually conjugate. To create a fully conjugate prior, we need to use a prior where $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are dependent on each other. We will use a joint distribution of the form $p(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = p(\boldsymbol{\mu} | \boldsymbol{\Sigma})p(\boldsymbol{\Sigma})$.

3. Using the language of directed graphical models, we see that $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ become dependent when conditioned on \mathcal{D} due to explaining away. See Figure 3.9(a).

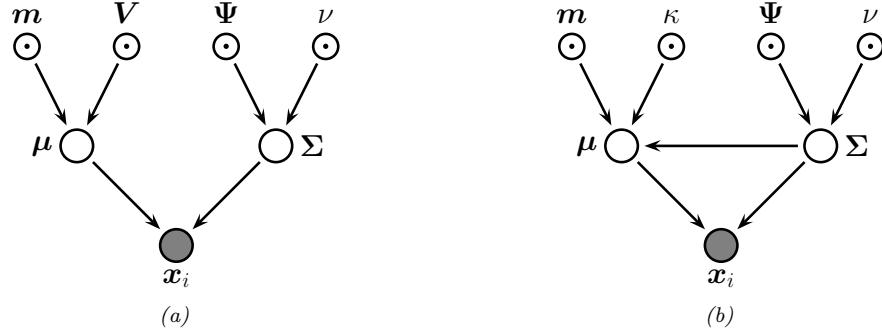


Figure 3.9: Graphical models representing different kinds of assumptions about the parameter priors. (a) A semi-conjugate prior for a Gaussian. (b) A conjugate prior for a Gaussian.

Looking at the form of the likelihood equation, Equation (3.151), we see that a natural conjugate prior has the form of a **normal-inverse-Wishart** or **NIW** distribution, defined as follows:

$$\text{NIW}(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \check{\boldsymbol{m}}, \check{\kappa}, \check{\nu}, \check{\boldsymbol{\Psi}}) \triangleq \mathcal{N}(\boldsymbol{\mu} | \check{\boldsymbol{m}}, \frac{1}{\check{\kappa}} \boldsymbol{\Sigma}) \times \text{IW}(\boldsymbol{\Sigma} | \check{\boldsymbol{\Psi}}^{-1}, \check{\nu}) \quad (3.153)$$

$$\begin{aligned} &= \frac{1}{Z_{\text{NIW}}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp \left(-\frac{\check{\kappa}}{2} (\boldsymbol{\mu} - \check{\boldsymbol{m}})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - \check{\boldsymbol{m}}) \right) \\ &\times |\boldsymbol{\Sigma}|^{-\frac{\check{\nu}+D+1}{2}} \exp \left(-\frac{1}{2} \text{tr}(\check{\boldsymbol{\Psi}} \boldsymbol{\Sigma}^{-1}) \right) \end{aligned} \quad (3.154)$$

where the normalization constant is given by

$$Z_{\text{NIW}} \triangleq 2^{\check{\nu}D/2} \Gamma_D(\check{\nu}/2) (2\pi/\check{\kappa})^{D/2} |\check{\boldsymbol{\Psi}}|^{\check{\nu}/2} \quad (3.155)$$

The parameters of the NIW can be interpreted as follows: $\check{\boldsymbol{m}}$ is our prior mean for $\boldsymbol{\mu}$, and $\check{\kappa}$ is how strongly we believe this prior; $\check{\boldsymbol{\Psi}}$ is (proportional to) our prior mean for $\boldsymbol{\Sigma}$, and $\check{\nu}$ is how strongly we believe this prior.⁴

Posterior

To derive the posterior, let us first rewrite the scatter matrix as follows:

$$\mathbf{S} = \mathbf{Y}^T \mathbf{Y} - \frac{1}{N} \left(\sum_{n=1}^N \mathbf{y}_n \right) \left(\sum_{n=1}^N \mathbf{y}_n \right)^T = \mathbf{Y}^T \mathbf{Y} - N \bar{\mathbf{y}} \bar{\mathbf{y}}^T \quad (3.156)$$

where $\mathbf{Y}^T \mathbf{Y} = \sum_{n=1}^N \mathbf{y}_n \mathbf{y}_n^T$ is the **sum of squares** matrix.

4. Note that our uncertainty in the mean is proportional to the covariance. In particular, if we believe that the variance is large, then our uncertainty in $\boldsymbol{\mu}$ must be large too. This makes sense intuitively, since if the data has large spread, it will be hard to pin down its mean.

Now we can multiply the likelihood and the prior to give

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D}) \propto |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp\left(-\frac{N}{2}(\boldsymbol{\mu} - \bar{\mathbf{y}})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - \bar{\mathbf{y}})\right) \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Sigma}^{-1}\mathbf{S})\right) \quad (3.157)$$

$$\times |\boldsymbol{\Sigma}|^{-\frac{\nu+D+2}{2}} \exp\left(-\frac{\kappa}{2}(\boldsymbol{\mu} - \tilde{\mathbf{m}})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - \tilde{\mathbf{m}})\right) \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Sigma}^{-1}\tilde{\Psi})\right) \quad (3.158)$$

$$= |\boldsymbol{\Sigma}|^{-(N+\nu+D+2)/2} \exp\left(-\frac{1}{2}\text{tr}(\boldsymbol{\Sigma}^{-1}\mathbf{M})\right) \quad (3.159)$$

where

$$\mathbf{M} \triangleq N(\boldsymbol{\mu} - \bar{\mathbf{y}})(\boldsymbol{\mu} - \bar{\mathbf{y}})^\top + \kappa(\boldsymbol{\mu} - \tilde{\mathbf{m}})(\boldsymbol{\mu} - \tilde{\mathbf{m}})^\top + \mathbf{S} + \tilde{\Psi} \quad (3.160)$$

$$= (\kappa + N)\boldsymbol{\mu}\boldsymbol{\mu}^\top - \boldsymbol{\mu}(\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})^\top - (\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})\boldsymbol{\mu}^\top + \kappa\tilde{\mathbf{m}}\tilde{\mathbf{m}}^\top + \mathbf{Y}^\top\mathbf{Y} + \tilde{\Psi} \quad (3.161)$$

We can simplify the \mathbf{M} matrix as follows:

$$(\kappa + N)\boldsymbol{\mu}\boldsymbol{\mu}^\top - \boldsymbol{\mu}(\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})^\top - (\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})\boldsymbol{\mu}^\top \quad (3.162)$$

$$= (\kappa + N) \left(\boldsymbol{\mu} - \frac{\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}}}{\kappa + N} \right) \left(\boldsymbol{\mu} - \frac{\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}}}{\kappa + N} \right)^\top \quad (3.163)$$

$$- \frac{(\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})(\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}})^\top}{\kappa + N} \quad (3.164)$$

$$= \hat{\kappa}(\boldsymbol{\mu} - \hat{\mathbf{m}})(\boldsymbol{\mu} - \hat{\mathbf{m}})^\top - \hat{\kappa}\hat{\mathbf{m}}\hat{\mathbf{m}}^\top \quad (3.165)$$

Hence we can rewrite the posterior as follows:

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D}) \propto |\boldsymbol{\Sigma}|^{(\hat{\nu}+D+2)/2} \exp\left(-\frac{1}{2}\text{tr}\left[\boldsymbol{\Sigma}^{-1}\left(\hat{\kappa}(\boldsymbol{\mu} - \hat{\mathbf{m}})(\boldsymbol{\mu} - \hat{\mathbf{m}})^\top + \hat{\Psi}\right)\right]\right) \quad (3.166)$$

$$= \text{NIW}(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \hat{\mathbf{m}}, \hat{\kappa}, \hat{\nu}, \hat{\Psi}) \quad (3.167)$$

where

$$\hat{\mathbf{m}} = \frac{\kappa\tilde{\mathbf{m}} + N\bar{\mathbf{y}}}{\hat{\kappa}} = \frac{\kappa}{\kappa + N}\tilde{\mathbf{m}} + \frac{N}{\kappa + N}\bar{\mathbf{y}} \quad (3.168)$$

$$\hat{\kappa} = \kappa + N \quad (3.169)$$

$$\hat{\nu} = \nu + N \quad (3.170)$$

$$\hat{\Psi} = \tilde{\Psi} + \mathbf{S} + \frac{\kappa N}{\kappa + N}(\bar{\mathbf{y}} - \tilde{\mathbf{m}})(\bar{\mathbf{y}} - \tilde{\mathbf{m}})^\top \quad (3.171)$$

$$= \tilde{\Psi} + \mathbf{Y}^\top\mathbf{Y} + \kappa\tilde{\mathbf{m}}\tilde{\mathbf{m}}^\top - \hat{\kappa}\hat{\mathbf{m}}\hat{\mathbf{m}}^\top \quad (3.172)$$

This result is actually quite intuitive: the posterior mean $\hat{\mathbf{m}}$ is a convex combination of the prior mean and the MLE; the posterior scatter matrix $\hat{\Psi}$ is the prior scatter matrix $\tilde{\Psi}$ plus the empirical scatter matrix \mathbf{S} plus an extra term due to the uncertainty in the mean (which creates its own virtual scatter matrix); and the posterior confidence factors $\hat{\kappa}$ and $\hat{\nu}$ are both incremented by the size of the data we condition on.

Posterior marginals

We have computed the joint posterior

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D}) = \mathcal{N}(\boldsymbol{\mu} | \boldsymbol{\Sigma}, \mathcal{D}) p(\boldsymbol{\Sigma} | \mathcal{D}) = \mathcal{N}(\boldsymbol{\mu} | \widehat{\boldsymbol{m}}, \frac{1}{\widehat{\kappa}} \boldsymbol{\Sigma}) \text{IW}(\boldsymbol{\Sigma} | \widehat{\boldsymbol{\Psi}}^{-1}, \widehat{\nu}) \quad (3.173)$$

We now discuss how to compute the posterior marginals, $p(\boldsymbol{\Sigma} | \mathcal{D})$ and $p(\boldsymbol{\mu} | \mathcal{D})$.

It is easy to see that the posterior marginal for $\boldsymbol{\Sigma}$ is

$$p(\boldsymbol{\Sigma} | \mathcal{D}) = \int p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D}) d\boldsymbol{\mu} = \text{IW}(\boldsymbol{\Sigma} | \widehat{\boldsymbol{\Psi}}^{-1}, \widehat{\nu}) \quad (3.174)$$

For the mean, one can show that

$$p(\boldsymbol{\mu} | \mathcal{D}) = \int p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathcal{D}) d\boldsymbol{\Sigma} = \mathcal{T}(\boldsymbol{\mu} | \widehat{\boldsymbol{\mu}}, \frac{\widehat{\boldsymbol{\Psi}}}{\widehat{\kappa} \widehat{\nu}'}, \widehat{\nu}') \quad (3.175)$$

where $\widehat{\nu}' \triangleq \widehat{\nu} - D + 1$. Intuitively this result follows because $p(\boldsymbol{\mu} | \mathcal{D})$ is an infinite mixture of Gaussians, where each mixture component has a value of $\boldsymbol{\Sigma}$ drawn from the IW distribution; by mixing these altogether, we induce a Student distribution, which has heavier tails than a single Gaussian.

Posterior mode

The maximum a posteriori (MAP) estimate of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ is the mode of the posterior NIW distribution with density

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \mathbf{Y}) = \mathcal{N}(\boldsymbol{\mu} | \widehat{\boldsymbol{\mu}}, \widehat{\kappa}^{-1} \boldsymbol{\Sigma}) \text{IW}(\boldsymbol{\Sigma} | \widehat{\boldsymbol{\Psi}}^{-1}, \widehat{\nu}) \quad (3.176)$$

To find the mode, we firstly notice that $\boldsymbol{\mu}$ only appears in the conditional distribution $\mathcal{N}(\boldsymbol{\mu} | \widehat{\boldsymbol{\mu}}, \widehat{\kappa}^{-1} \boldsymbol{\Sigma})$, and the mode of this normal distribution equals its mean, i.e., $\boldsymbol{\mu} = \widehat{\boldsymbol{\mu}}$. Also notice that this holds for any choice of $\boldsymbol{\Sigma}$. So we can plug $\boldsymbol{\mu} = \widehat{\boldsymbol{\mu}}$ in Equation (3.176) and derive the mode of $\boldsymbol{\Sigma}$. Notice that

$$-2 * \log p(\boldsymbol{\mu} = \widehat{\boldsymbol{\mu}}, \boldsymbol{\Sigma} | \mathbf{Y}) = (\widehat{\nu} + D + 2) \log(|\boldsymbol{\Sigma}|) + \text{tr}(\widehat{\boldsymbol{\Psi}} \boldsymbol{\Sigma}^{-1}) + c \quad (3.177)$$

where c is a constant irrelevant to $\boldsymbol{\Sigma}$. We then take the derivative over $\boldsymbol{\Sigma}$:

$$\frac{\partial \log p(\boldsymbol{\mu} = \widehat{\boldsymbol{\mu}}, \boldsymbol{\Sigma} | \mathbf{Y})}{\partial \boldsymbol{\Sigma}} = (\widehat{\nu} + D + 2) \boldsymbol{\Sigma}^{-1} - \boldsymbol{\Sigma}^{-1} \widehat{\boldsymbol{\Psi}} \boldsymbol{\Sigma}^{-1} \quad (3.178)$$

By setting the derivative to 0 and solving for $\boldsymbol{\Sigma}$, we see that $(\widehat{\nu} + D + 2)^{-1} \widehat{\boldsymbol{\Psi}}$ is the matrix that maximizes Equation (3.177). By checking that $\widehat{\boldsymbol{\Psi}}$ is a positive definite matrix, we conclude that $\widehat{\boldsymbol{\Psi}}$ is the MAP estimate of the covariance matrix $\boldsymbol{\Sigma}$.

In conclusion, the MAP estimate of $\{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ are

$$\widehat{\boldsymbol{\mu}} = \frac{\widehat{\kappa} \widehat{\boldsymbol{\mu}} + N \bar{\mathbf{y}}}{\widehat{\kappa} + N} \quad (3.179)$$

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{\widehat{\nu} + D + 2} \widehat{\boldsymbol{\Psi}} \quad (3.180)$$

Posterior predictive

We now discuss how to predict future data by integrating out the parameters. If $\mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $(\boldsymbol{\mu}, \boldsymbol{\Sigma}|\mathcal{D}) \sim \text{NIW}(\hat{\boldsymbol{m}}, \hat{\kappa}, \hat{\nu}, \hat{\boldsymbol{\Psi}})$, then one can show that the posterior predictive distribution, for a single observation vector, is as follows:

$$p(\mathbf{y}|\mathcal{D}) = \int \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \text{NIW}(\boldsymbol{\mu}, \boldsymbol{\Sigma} | \hat{\boldsymbol{m}}, \hat{\kappa}, \hat{\nu}, \hat{\boldsymbol{\Psi}}) d\boldsymbol{\mu} d\boldsymbol{\Sigma} \quad (3.181)$$

$$= \mathcal{T}(\mathbf{y} | \hat{\boldsymbol{m}}, \frac{\hat{\boldsymbol{\Psi}}(\hat{\kappa}+1)}{\hat{\kappa}\hat{\nu}'}, \hat{\nu}') \quad (3.182)$$

where $\hat{\nu}' = \hat{\nu} - D + 1$.

3.4.5 The exponential family model

We have seen that exact Bayesian analysis is considerably simplified if the prior is conjugate to the likelihood. Since the posterior must have the same form as the prior, and hence the same number of parameters, the likelihood function must have fixed-sized sufficient statistics, so that we can write $p(\mathcal{D}|\boldsymbol{\theta}) = p(\mathbf{s}(\mathcal{D})|\boldsymbol{\theta})$. This suggests that the only family of distributions for which conjugate priors exist is the exponential family, a result proved in [DY79].⁵ In the sections below, we show how to perform conjugate analysis for a generic exponential family model.

3.4.5.1 Likelihood

Recall that the likelihood of the exponential family is given by

$$p(\mathcal{D}|\boldsymbol{\eta}) = h(\mathcal{D}) \exp(\boldsymbol{\eta}^\top \mathbf{s}(\mathcal{D}) - N A(\boldsymbol{\eta})) \quad (3.183)$$

where $\mathbf{s}(\mathcal{D}) = \sum_{n=1}^N \mathbf{s}(\mathbf{x}_n)$ and $h(\mathcal{D}) \triangleq \prod_{n=1}^N h(\mathbf{x}_n)$.

3.4.5.2 Prior

Let us write the prior in a form that mirrors the likelihood:

$$p(\boldsymbol{\eta} | \tilde{\boldsymbol{\tau}}, \tilde{\nu}) = \frac{1}{Z(\tilde{\boldsymbol{\tau}}, \tilde{\nu})} \exp(\tilde{\boldsymbol{\tau}}^\top \boldsymbol{\eta} - \tilde{\nu} A(\boldsymbol{\eta})) \quad (3.184)$$

where $\tilde{\nu}$ is the strength of the prior, and $\tilde{\boldsymbol{\tau}} / \tilde{\nu}$ is the prior mean, and $Z(\tilde{\boldsymbol{\tau}}, \tilde{\nu})$ is a normalizing factor. The parameters $\tilde{\boldsymbol{\tau}}$ can be derived from **virtual samples** representing our prior beliefs.

5. There are some exceptions. For example, the uniform distribution $\text{Unif}(x|0, \theta)$ has finite sufficient statistics $(N, m = \max_i x_i)$, as discussed in Section 2.4.2.6; hence this distribution has a conjugate prior, namely the Pareto distribution (Section 2.2.3.5), $p(\theta) = \text{Pareto}(\theta|\theta_0, \kappa)$, yielding the posterior $p(\theta|\mathbf{x}) = \text{Pareto}(\max(\theta_0, m), \kappa + N)$.

3.4.5.3 Posterior

The posterior is given by

$$p(\boldsymbol{\eta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\eta})p(\boldsymbol{\eta})}{p(\mathcal{D})} \quad (3.185)$$

$$= \frac{h(\mathcal{D})}{Z(\bar{\tau}, \bar{\nu})p(\mathcal{D})} \exp((\bar{\tau} + \mathbf{s}(\mathcal{D}))^\top \boldsymbol{\eta} - (\bar{\nu} + N)A(\boldsymbol{\eta})) \quad (3.186)$$

$$= \frac{1}{Z(\hat{\tau}, \hat{\nu})} \exp(\hat{\tau}^\top \boldsymbol{\eta} - \hat{\nu} A(\boldsymbol{\eta})) \quad (3.187)$$

where

$$\hat{\tau} = \bar{\tau} + \mathbf{s}(\mathcal{D}) \quad (3.188)$$

$$\hat{\nu} = \bar{\nu} + N \quad (3.189)$$

$$Z(\hat{\tau}, \hat{\nu}) = \frac{Z(\bar{\tau}, \bar{\nu})}{h(\mathcal{D})} p(\mathcal{D}) \quad (3.190)$$

We see that this has the same form as the prior, but where we update the sufficient statistics and the sample size.

The posterior mean is given by a convex combination of the prior mean and the empirical mean (which is the MLE):

$$\mathbb{E}[\boldsymbol{\eta}|\mathcal{D}] = \frac{\hat{\tau}}{\hat{\nu}} = \frac{\bar{\tau} + \mathbf{s}(\mathcal{D})}{\bar{\nu} + N} = \frac{\bar{\nu}}{\bar{\nu} + N} \frac{\bar{\tau}}{\bar{\nu}} + \frac{N}{\bar{\nu} + N} \frac{\mathbf{s}(\mathcal{D})}{N} \quad (3.191)$$

$$= \lambda \mathbb{E}[\boldsymbol{\eta}] + (1 - \lambda) \hat{\boldsymbol{\eta}}_{\text{mle}} \quad (3.192)$$

where $\lambda = \frac{\bar{\nu}}{\bar{\nu} + N}$.

3.4.5.4 Marginal likelihood

From Equation (3.190) we see that the marginal likelihood is given by

$$p(\mathcal{D}) = \frac{Z(\hat{\tau}, \hat{\nu})h(\mathcal{D})}{Z(\bar{\tau}, \bar{\nu})} \quad (3.193)$$

See Section 3.2.1.9 for a detailed example in the case of the beta-Bernoulli model.

3.4.5.5 Posterior predictive density

We now derive the predictive density for future observables $\mathcal{D}' = (\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_{N'})$ given past data $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$:

$$p(\mathcal{D}'|\mathcal{D}) = \int p(\mathcal{D}'|\boldsymbol{\eta})p(\boldsymbol{\eta}|\mathcal{D})d\boldsymbol{\eta} \quad (3.194)$$

$$= \int h(\mathcal{D}') \exp(\boldsymbol{\eta}^\top \mathbf{s}(\mathcal{D}') - N' A(\boldsymbol{\eta})) \frac{1}{Z(\hat{\tau}, \hat{\nu})} \exp(\boldsymbol{\eta}^\top \hat{\tau} - \hat{\nu} A(\boldsymbol{\eta})) d\boldsymbol{\eta} \quad (3.195)$$

$$= h(\mathcal{D}') \frac{Z(\bar{\tau} + \mathbf{s}(\mathcal{D}) + \mathbf{s}(\mathcal{D}'), \bar{\nu} + N + N')}{Z(\bar{\tau} + \mathbf{s}(\mathcal{D}), \bar{\nu} + N)} \quad (3.196)$$

3.4.5.6 Example: Bernoulli distribution

As a simple example, let us revisit the Beta-Bernoulli model in our new notation.

The likelihood is given by

$$p(\mathcal{D}|\theta) = (1-\theta)^N \exp\left(\log\left(\frac{\theta}{1-\theta}\right)\sum_i x_n\right) \quad (3.197)$$

Hence the conjugate prior is given by

$$p(\theta|\nu_0, \tau_0) \propto (1-\theta)^{\nu_0} \exp\left(\log\left(\frac{\theta}{1-\theta}\right)\tau_0\right) \quad (3.198)$$

$$= \theta^{\tau_0} (1-\theta)^{\nu_0 - \tau_0} \quad (3.199)$$

If we define $\alpha = \tau_0 + 1$ and $\beta = \nu_0 - \tau_0 + 1$, we see that this is a beta distribution.

We can derive the posterior as follows, where $s = \sum_i \mathbb{I}(x_i = 1)$ is the sufficient statistic:

$$p(\theta|\mathcal{D}) \propto \theta^{\tau_0+s} (1-\theta)^{\nu_0 - \tau_0 + n - s} \quad (3.200)$$

$$= \theta^{\tau_n} (1-\theta)^{\nu_n - \tau_n} \quad (3.201)$$

We can derive the posterior predictive distribution as follows. Assume $p(\theta) = \text{Beta}(\theta|\alpha, \beta)$, and let $s = s(\mathcal{D})$ be the number of heads in the past data. We can predict the probability of a given sequence of future heads, $\mathcal{D}' = (\tilde{x}_1, \dots, \tilde{x}_m)$, with sufficient statistic $s' = \sum_{n=1}^m \mathbb{I}(\tilde{x}_i = 1)$, as follows:

$$p(\mathcal{D}'|\mathcal{D}) = \int_0^1 p(\mathcal{D}'|\theta|\text{Beta}(\theta|\alpha_n, \beta_n)) d\theta \quad (3.202)$$

$$= \frac{\Gamma(\alpha_n + \beta_n)}{\Gamma(\alpha_n)\Gamma(\beta_n)} \int_0^1 \theta^{\alpha_n + t' - 1} (1-\theta)^{\beta_n + m - t' - 1} d\theta \quad (3.203)$$

$$= \frac{\Gamma(\alpha_n + \beta_n)}{\Gamma(\alpha_n)\Gamma(\beta_n)} \frac{\Gamma(\alpha_{n+m})\Gamma(\beta_{n+m})}{\Gamma(\alpha_{n+m})\Gamma(\beta_{n+m})} \quad (3.204)$$

where

$$\alpha_{n+m} = \alpha_n + s' = \alpha + s + s' \quad (3.205)$$

$$\beta_{n+m} = \beta_n + (m - s') = \beta + (n - s) + (m - s') \quad (3.206)$$

3.4.6 Beyond conjugate priors

We have seen various examples of conjugate priors, all of which have come from the exponential family (see Section 2.4). These priors have the advantages of being easy to interpret (in terms of sufficient statistics from a virtual prior dataset), and being easy to compute with. However, for most models, there is no prior in the exponential family that is conjugate to the likelihood. Furthermore, even where there is a conjugate prior, the assumption of conjugacy may be too limiting. Therefore in the sections below, we briefly discuss various other kinds of priors. (We defer the question of posterior inference with these priors until Section 7.1, where we discuss algorithmic issues, since we can no longer use closed-form solutions when the prior is not conjugate.)

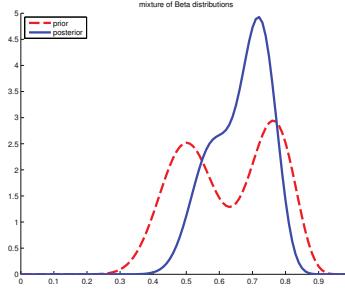


Figure 3.10: A mixture of two Beta distributions. Generated by `mixbetademo.ipynb`.

3.4.6.1 Mixtures of conjugate priors

In this section, we show how we can create a **mixture of conjugate priors** for increased modeling flexibility. Fortunately, the resulting mixture prior is still conjugate.

As an example, suppose we want to predict the outcome of a coin toss at a casino, and we believe that the coin may be fair, but it may also be biased towards heads. This prior cannot be represented by a beta distribution. Fortunately, it can be represented as a mixture of beta distributions. For example, we might use

$$p(\theta) = 0.5 \text{ Beta}(\theta|20, 20) + 0.5 \text{ Beta}(\theta|30, 10) \quad (3.207)$$

If θ comes from the first distribution, the coin is fair, but if it comes from the second, it is biased towards heads.

We can represent a mixture by introducing a latent indicator variable h , where $h = k$ means that θ comes from mixture component k . The prior has the form

$$p(\theta) = \sum_k p(h = k)p(\theta|h = k) \quad (3.208)$$

where each $p(\theta|h = k)$ is conjugate, and $p(h = k)$ are called the (prior) mixing weights. One can show that the posterior can also be written as a mixture of conjugate distributions as follows:

$$p(\theta|\mathcal{D}) = \sum_k p(h = k|\mathcal{D})p(\theta|\mathcal{D}, h = k) \quad (3.209)$$

where $p(h = k|\mathcal{D})$ are the posterior mixing weights given by

$$p(h = k|\mathcal{D}) = \frac{p(h = k)p(\mathcal{D}|h = k)}{\sum_{k'} p(h = k')p(\mathcal{D}|h = k')} \quad (3.210)$$

Here the quantity $p(\mathcal{D}|h = k)$ is the marginal likelihood for mixture component k (see Section 3.2.1.9).

Returning to our example above, if we have the prior in Equation (3.207), and we observe $N_1 = 20$ heads and $N_0 = 10$ tails, then, using Equation (3.31), the posterior becomes

$$p(\theta|\mathcal{D}) = 0.346 \text{ Beta}(\theta|40, 30) + 0.654 \text{ Beta}(\theta|50, 20) \quad (3.211)$$

See Figure 3.10 for an illustration.

We can compute the posterior probability that the coin is biased towards heads as follows:

$$\Pr(\theta > 0.5 | \mathcal{D}) = \sum_k \Pr(\theta > 0.5 | \mathcal{D}, h = k) p(h = k | \mathcal{D}) = 0.9604 \quad (3.212)$$

If we just used a single Beta(20,20) prior, we would get a slightly smaller value of $\Pr(\theta > 0.5 | \mathcal{D}) = 0.8858$. So if we were “suspicious” initially that the casino might be using a biased coin, our fears would be confirmed more quickly than if we had to be convinced starting with an open mind.

3.4.6.2 Robust (heavy-tailed) priors

The assessment of the influence of the prior on the posterior is called **sensitivity analysis**, or **robustness analysis**. There are many ways to create **robust priors**. (see e.g., [IR00]). Here we consider a simple approach, namely the use of a heavy-tailed distribution.

To motivate this, let us consider an example from [Ber85a, p7]. Suppose $x \sim \mathcal{N}(\theta, 1)$. We observe that $x = 5$ and we want to estimate θ . The MLE is of course $\hat{\theta} = 5$, which seems reasonable. The posterior mean under a uniform prior is also $\bar{\theta} = 5$. But now suppose we know that the prior median is 0, and that there is 25% probability that θ lies in any of the intervals $(-\infty, -1)$, $(-1, 0)$, $(0, 1)$, $(1, \infty)$. Let us also assume the prior is smooth and unimodal.

One can show that a Gaussian prior of the form $\mathcal{N}(\theta | 0, 2.19^2)$ satisfies these prior constraints. But in this case the posterior mean is given by 3.43, which doesn’t seem very satisfactory. An alternative distribution that captures the same prior information is the Cauchy prior $\mathcal{T}_1(\theta | 0, 1)$. With this prior, we find (using numerical method integration: see [robust_prior_demo.ipynb](#) for the code) that the posterior mean is about 4.6, which seems much more reasonable. In general, priors with heavy tails tend to give results which are more sensitive to the data, which is usually what we desire.

Heavy-tailed priors are usually not conjugate. However, we can often approximate a heavy-tailed prior by using a (possibly infinite) mixture of conjugate priors. For example, in Section 28.2.3, we show that the Student distribution (of which the Cauchy is a special case) can be written as an infinite mixture of Gaussians, where the mixing weights come from a gamma distribution. This is an example of a hierarchical prior; see Section 3.6 for details.

3.4.6.3 Priors for scalar variances

In this section, we discuss some commonly used priors for variance parameters. Such priors play an important role in determining how much regularization a model exhibits. For example, consider a linear regression model, $p(y | \mathbf{x}, \mathbf{w}, \sigma^2) = \mathcal{N}(y | \mathbf{w}^\top \mathbf{x}, \sigma^2)$. Suppose we use a Gaussian prior on the weights, $p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \tau^2 \mathbf{I})$. The value of τ^2 (relative to σ^2) plays a role similar to the strength of an ℓ_2 -regularization term in ridge regression. In the Bayesian setting, we need to ensure we use sensible priors for the variance parameters, τ^2 and σ^2 . This becomes even more important when we discuss hierarchical models, in Section 3.6.

We start by considering the simple problem of inferring a variance parameter σ^2 from a Gaussian likelihood with known mean, as in Section 3.4.3.2. The uninformative prior is $p(\sigma^2) = \text{IG}(\sigma^2 | 0, 0)$, which is improper, meaning it does not integrate to 1. This is fine as long as the posterior is proper. This will be the case if the prior is on the variance of the noise of $N \geq 2$ observable variables.

Unfortunately the posterior is not proper, even if $N \rightarrow \infty$, if we use this prior for the variance of the (non observable) weights in a regression model [Gel06; PS12], as we discuss in Section 3.6.

One solution to this is to use a **weakly informative** proper prior such as $\text{IG}(\epsilon, \epsilon)$ for small ϵ . However, this turns out to not work very well, for reasons that are explained in [Gel06; PS12]. Instead, it is recommended to use other priors, such as uniform, exponential, half-normal, half-Student- t , or half-Cauchy; all of these are bounded below by 0, and just require 1 or 2 hyperparameters. (The term “half” refers to the fact that the distribution is “folded over” onto itself on the positive side of the real axis.)

3.4.6.4 Priors for covariance matrices

The conjugate prior for a covariance matrix is the inverse Wishart (Section 2.2.5.6). However, it can be hard to set the parameters for this in an uninformative way. One approach, discussed in [HW13], is to use a scale mixture of inverse Wisharts, where the scaling parameters have inverse gamma distributions. It is possible to choose shape and scale parameters to ensure that all the correlation parameters have uniform $(-1, 1)$ marginals, and all the standard deviations have half-Student distributions.

Unfortunately, the Wishart distribution has heavy tails, which can lead to poor performance when used in a sampling algorithm.⁶ A more common approach, following Equation (3.213), is to represent the $D \times D$ covariance matrix Σ in terms of a product of the marginal standard deviations, $\sigma = (\sigma_1, \dots, \sigma_D)$, and the $D \times D$ correlation matrix \mathbf{R} , as follows:

$$\Sigma = \text{diag}(\sigma) \mathbf{R} \text{diag}(\sigma) \quad (3.213)$$

For example, if $D = 2$, we have

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \quad (3.214)$$

We can put a factored prior on the standard deviations, following the recommendations of Section 3.4.6.3. For example,

$$p(\sigma) = \prod_{d=1}^D \text{Expon}(\sigma_d | 1) \quad (3.215)$$

For the correlation matrix, it is common to use as a prior the **LKJ distribution**, named after the authors of [LKJ09]. This has the form

$$\text{LKJ}(\mathbf{R}|\eta) \propto |\mathbf{R}|^{\eta-1} \quad (3.216)$$

so it only has one free parameter. When $\eta = 1$, it is a uniform prior; when $\eta = 2$, it is a “weakly regularizing” prior, that encourages small correlations (close to 0). See Figure 3.11 for a plot.

In practice, it is more common to define \mathbf{R} in terms of its Cholesky decomposition, $\mathbf{R} = \mathbf{L}\mathbf{L}^\top$, where \mathbf{L} is an unconstrained lower triangular matrix. We then represent the prior using

$$\text{LKJchol}(\mathbf{L}|\eta) \propto |\mathbf{L}|^{-\eta-1} \quad (3.217)$$

6. See comments from Michael Betancourt at <https://github.com/pymc-devs/pymc/issues/538>.

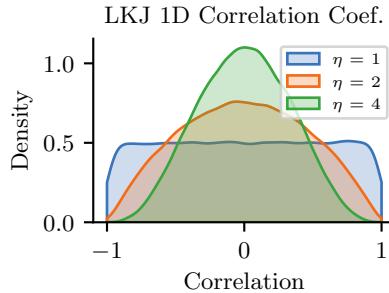


Figure 3.11: Distribution on the correlation coefficient ρ induced by a 2d LKJ distribution with varying parameter. Adapted from Figure 14.3 of [McE20]. Generated by [lkj_1d.ipynb](#).

3.5 Noninformative priors

When we have little or no domain specific knowledge, it is desirable to use an **uninformative**, **noninformative**, or **objective** priors, to “let the data speak for itself”. Unfortunately, there is no unique way to define such priors, and they all encode some kind of knowledge. It is therefore better to use the term **diffuse prior**, **minimally informative prior**, or **default prior**.

In the sections below, we briefly mention some common approaches for creating default priors. For further details, see e.g., [KW96] and the Stan website.⁷

3.5.1 Maximum entropy priors

A natural way to define an uninformative prior is to use one that has **maximum entropy**, since it makes the least commitments to any particular value in the state space (see Section 5.2 for a discussion of entropy). This is a formalization of Laplace’s **principle of insufficient reason**, in which he argued that if there is no reason to prefer one prior over another, we should pick a “flat” one.

For example, in the case of a Bernoulli distribution with rate $\theta \in [0, 1]$, the maximum entropy prior is the uniform distribution, $p(\theta) = \text{Beta}(\theta|1, 1)$, which makes intuitive sense.

However, in some cases we know something about our random variable $\boldsymbol{\theta}$, and we would like our prior to match these constraints, but otherwise be maximally entropic. More precisely, suppose we want to find a distribution $p(\boldsymbol{\theta})$ with maximum entropy, subject to the constraints that the expected values of certain features or functions $f_k(\boldsymbol{\theta})$ match some known quantities F_k . This is called a **maxent prior**. In Section 2.4.7, we show that such distributions must belong to the exponential family (Section 2.4).

For example, suppose $\theta \in \{1, 2, \dots, 10\}$, and let $p_c = p(\theta = c)$ be the corresponding prior. Suppose we know that the prior mean is 1.5. We can encode this using the following constraint

$$\mathbb{E}[f_1(\theta)] = \mathbb{E}[\theta] = \sum_c c p_c = 1.5 \tag{3.218}$$

7. <https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations>.

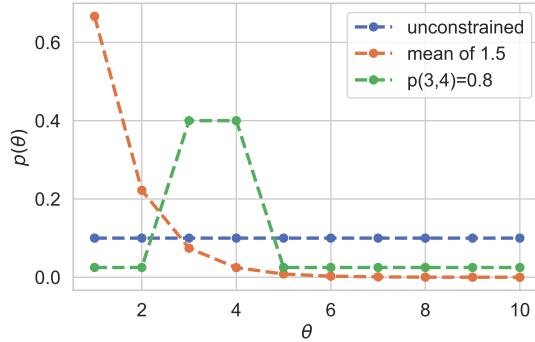


Figure 3.12: Illustration of 3 different maximum entropy priors. Adapted from Figure 1.10 of [MKL11]. Generated by [maxent_priors.ipynb](#).

In addition, we have the constraint $\sum_c p_c = 1$. Thus we need to solve the following optimization problem:

$$\min_{\mathbf{p}} \mathbb{H}(\mathbf{p}) \quad \text{s.t.} \quad \sum_c c p_c = 1.5, \quad \sum_c p_c = 1.0 \quad (3.219)$$

This gives the decaying exponential curve in Figure 3.12. Now suppose we know that θ is either 3 or 4 with probability 0.8. We can encode this using

$$\mathbb{E}[f_1(\theta)] = \mathbb{E}[\mathbb{I}(\theta \in \{3, 4\})] = \Pr(\theta \in \{3, 4\}) = 0.8 \quad (3.220)$$

This gives the inverted U-curve in Figure 3.12. We note that this distribution is flat in as many places as possible.

3.5.2 Jeffreys priors

Let θ be a random variable with prior $p_\theta(\theta)$, and let $\phi = f(\theta)$ be some invertible transformation of θ . We want to choose a prior that is **invariant** to this function f , so that the posterior does not depend on how we parameterize the model.

For example, consider a Bernoulli distribution with rate parameter θ . Suppose Alice uses a binomial likelihood with data \mathcal{D} , and computes $p(\theta|\mathcal{D})$. Now suppose Bob uses the same likelihood and data, but parameterizes the model in terms of the odds parameter, $\phi = \frac{\theta}{1-\theta}$. He converts Alice's prior to $p(\phi)$ using the change of variables formula, and then computes $p(\phi|\mathcal{D})$. If he then converts back to the θ parameterization, he should get the same result as Alice.

We can achieve this goal if provided we use a **Jeffreys prior**, named after Harold Jeffreys.⁸ In 1d, the Jeffreys prior is given by $p(\theta) \propto \sqrt{F(\theta)}$, where F is the Fisher information (Section 3.3.4).

8. Harold Jeffreys, 1891–1989, was an English mathematician, statistician, geophysicist, and astronomer. He is not to be confused with Richard Jeffrey, a philosopher who advocated the subjective interpretation of probability [Jef04].

In multiple dimensions, the Jeffreys prior has the form $p(\boldsymbol{\theta}) \propto \sqrt{\det \mathbf{F}(\boldsymbol{\theta})}$, where \mathbf{F} is the Fisher information matrix (Section 3.3.4).

To see why the Jeffreys prior is invariant to parameterization, consider the 1d case. Suppose $p_\theta(\theta) \propto \sqrt{F(\theta)}$. Using the change of variables, we can derive the corresponding prior for ϕ as follows:

$$p_\phi(\phi) = p_\theta(\theta) \left| \frac{d\theta}{d\phi} \right| \quad (3.221)$$

$$\propto \sqrt{F(\theta) \left(\frac{d\theta}{d\phi} \right)^2} = \sqrt{\mathbb{E} \left[\left(\frac{d \log p(x|\theta)}{d\theta} \right)^2 \right] \left(\frac{d\theta}{d\phi} \right)^2} \quad (3.222)$$

$$= \sqrt{\mathbb{E} \left[\left(\frac{d \log p(x|\theta)}{d\theta} \frac{d\theta}{d\phi} \right)^2 \right]} = \sqrt{\mathbb{E} \left[\left(\frac{d \log p(x|\phi)}{d\phi} \right)^2 \right]} \quad (3.223)$$

$$= \sqrt{F(\phi)} \quad (3.224)$$

Thus the prior distribution is the same whether we use the θ parameterization or the ϕ parameterization.

We give some examples of Jeffreys priors below.

3.5.2.1 Jeffreys prior for binomial distribution

Let us derive the Jeffreys prior for the binomial distribution using the rate parameterization θ . From Equation (3.57), we have

$$p(\theta) \propto \theta^{-\frac{1}{2}} (1-\theta)^{-\frac{1}{2}} = \frac{1}{\sqrt{\theta(1-\theta)}} \propto \text{Beta}(\theta | \frac{1}{2}, \frac{1}{2}) \quad (3.225)$$

Now consider the odds parameterization, $\phi = \theta/(1-\theta)$, so $\theta = \frac{\phi}{\phi+1}$. The likelihood becomes

$$p(x|\phi) \propto \left(\frac{\phi}{\phi+1} \right)^x \left(1 - \frac{\phi}{\phi+1} \right)^{n-x} = \phi^x (\phi+1)^{-x} (\phi+1)^{-n+x} = \phi^x (\phi+1)^{-n} \quad (3.226)$$

Thus the log likelihood is

$$\ell = x \log \phi - n \log \phi + 1 \quad (3.227)$$

The first and second derivatives are

$$\frac{d\ell}{d\phi} = \frac{x}{\phi} - \frac{n}{\phi+1} \quad (3.228)$$

$$\frac{d^2\ell}{d\phi^2} = -\frac{x}{\phi^2} + \frac{n}{(\phi+1)^2} \quad (3.229)$$

Since $\mathbb{E}[x] = n\theta = n\frac{\phi}{\phi+1}$, the Fisher information matrix is given by

$$F(\phi) = -\mathbb{E} \left[\frac{d^2\ell}{d\phi^2} \right] \frac{n}{\phi(\phi+1)} - \frac{n}{(\phi+1)^2} \quad (3.230)$$

$$= \frac{n(\phi+1) - n\phi}{\phi(\phi+1)^2} = \frac{n}{\phi(\phi+1)^2} \quad (3.231)$$

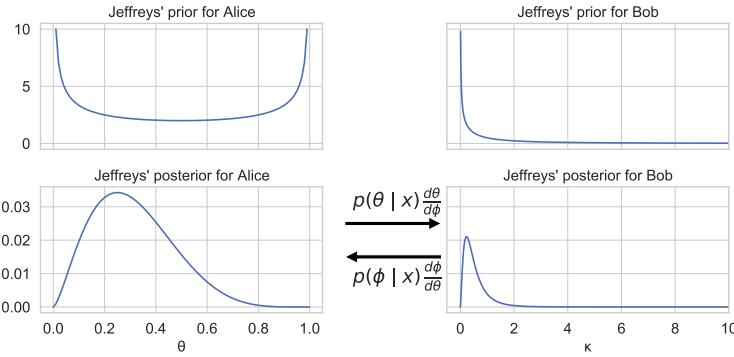


Figure 3.13: Illustration of Jeffreys prior for Alice (who uses the rate θ) and Bob (who uses the odds $\phi = \theta/(1 - \theta)$). Adapted from Figure 1.9 of [MKL11]. Generated by `jeffreys_prior_binomial.ipynb`.

Hence

$$p_\phi(\phi) \propto \phi^{-0.5} (1 + \phi)^{-1} \quad (3.232)$$

See Figure 3.13 for an illustration.

3.5.2.2 Jeffreys prior for multinomial distribution

For a categorical random variable with K states, one can show that the Jeffreys prior is given by

$$p(\boldsymbol{\theta}) \propto \text{Dir}(\boldsymbol{\theta} | \frac{1}{2}, \dots, \frac{1}{2}) \quad (3.233)$$

Note that this is different from the more obvious choices of $\text{Dir}(\frac{1}{K}, \dots, \frac{1}{K})$ or $\text{Dir}(1, \dots, 1)$.

3.5.2.3 Jeffreys prior for the mean and variance of a univariate Gaussian

Consider a 1d Gaussian $x \sim \mathcal{N}(\mu, \sigma^2)$ with both parameters unknown, so $\boldsymbol{\theta} = (\mu, \sigma)$. From Equation (3.62), the Fisher information matrix is

$$\mathbf{F}(\boldsymbol{\theta}) = \begin{pmatrix} 1/\sigma^2 & 0 \\ 0 & 2/\sigma^2 \end{pmatrix} \quad (3.234)$$

so $\sqrt{\det(\mathbf{F}(\boldsymbol{\theta}))} = \sqrt{\frac{2}{\sigma^4}}$. However, the standard Jeffreys uninformative prior for the Gaussian is defined as the product of independent uninformative priors (see [KW96]), i.e.,

$$p(\mu, \sigma^2) \propto p(\mu)p(\sigma^2) \propto 1/\sigma^2 \quad (3.235)$$

It turns out that we can emulate this prior with a conjugate NIX prior:

$$p(\mu, \sigma^2) = NI\chi^2(\mu, \sigma^2 | \mu_0 = 0, \kappa = 0, \nu = -1, \tilde{\sigma}^2 = 0) \quad (3.236)$$

This lets us easily reuse the results for conjugate analysis of the Gaussian in Section 3.4.3.3, as we showed in Section 3.4.3.4.

3.5.3 Invariant priors

If we have “objective” prior knowledge about a problem in the form of invariances, we may be able to encode this into a prior, as we show below.

3.5.3.1 Translation-invariant priors

A **location-scale family** is a family of probability distributions parameterized by a location μ and scale σ . If x is an rv in this family, then $y = a + bx$ is also an rv in the same family.

When inferring the location parameter μ , it is intuitively reasonable to want to use a **translation-invariant prior**, which satisfies the property that the probability mass assigned to any interval, $[A, B]$ is the same as that assigned to any other shifted interval of the same width, such as $[A - c, B - c]$. That is,

$$\int_{A-c}^{B-c} p(\mu) d\mu = \int_A^B p(\mu) d\mu \quad (3.237)$$

This can be achieved using

$$p(\mu) \propto 1 \quad (3.238)$$

since

$$\int_{A-c}^{B-c} 1 d\mu = (B - c) - (A - c) = (B - A) = \int_A^B 1 d\mu \quad (3.239)$$

This is the same as the Jeffreys prior for a Gaussian with unknown mean μ and fixed variance. This follows since $F(\mu) = 1/\sigma^2 \propto 1$, from Equation (3.62), and hence $p(\mu) \propto 1$.

3.5.3.2 Scale-invariant prior

When inferring the scale parameter σ , we may want to use a **scale-invariant prior**, which satisfies the property that the probability mass assigned to any interval $[A, B]$ is the same as that assigned to any other interval $[A/c, B/c]$, where $c > 0$. That is,

$$\int_{A/c}^{B/c} p(\sigma) d\sigma = \int_A^B p(\sigma) d\sigma \quad (3.240)$$

This can be achieved by using

$$p(\sigma) \propto 1/\sigma \quad (3.241)$$

since then

$$\int_{A/c}^{B/c} \frac{1}{\sigma} d\sigma = [\log \sigma]_{A/c}^{B/c} = \log(B/c) - \log(A/c) = \log(B) - \log(A) = \int_A^B \frac{1}{\sigma} d\sigma \quad (3.242)$$

This is the same as the Jeffreys prior for a Gaussian with fixed mean μ and unknown scale σ . This follows since $F(\sigma) = 2/\sigma^2$, from Equation (3.62), and hence $p(\sigma) \propto 1/\sigma$.

3.5.3.3 Learning invariant priors

Whenever we have knowledge of some kind of invariance we want our model to satisfy, we can use this to encode a corresponding prior. Sometimes this is done analytically (see e.g., [Rob07, Ch.9]). When this is intractable, it may be possible to learn invariant priors by solving a variational optimization problem (see e.g., [NS18]).

3.5.4 Reference priors

One way to define a noninformative prior is as a distribution which is maximally far from all possible posteriors, when averaged over datasets. This is the basic idea behind a **reference prior** [Ber05; BBS09]. More precisely, we say that $p(\boldsymbol{\theta})$ is a reference prior if it maximizes the expected KL divergence between posterior and prior:

$$p^*(\boldsymbol{\theta}) = \underset{p(\boldsymbol{\theta})}{\operatorname{argmax}} \int_{\mathcal{D}} p(\mathcal{D}) D_{\text{KL}}(p(\boldsymbol{\theta}|\mathcal{D}) \parallel p(\boldsymbol{\theta})) d\mathcal{D} \quad (3.243)$$

where $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$. This is the same as maximizing the mutual information $\mathbb{I}(\boldsymbol{\theta}, \mathcal{D})$.

We can eliminate the integral over datasets by noting that

$$\int p(\mathcal{D}) \int p(\boldsymbol{\theta}|\mathcal{D}) \log \frac{p(\boldsymbol{\theta}|\mathcal{D})}{p(\boldsymbol{\theta})} = \int p(\boldsymbol{\theta}) \int p(\mathcal{D}|\boldsymbol{\theta}) \log \frac{p(\mathcal{D}|\boldsymbol{\theta})}{p(\mathcal{D})} = \mathbb{E}_{\boldsymbol{\theta}} [D_{\text{KL}}(p(\mathcal{D}|\boldsymbol{\theta}) \parallel p(\mathcal{D}))] \quad (3.244)$$

where we used the fact that $\frac{p(\boldsymbol{\theta}|\mathcal{D})}{p(\boldsymbol{\theta})} = \frac{p(\mathcal{D}|\boldsymbol{\theta})}{p(\mathcal{D})}$.

One can show that, in 1d, the corresponding prior is equivalent to the Jeffreys prior. In higher dimensions, we can compute the reference prior for one parameter at a time, using the chain rule. However, this can become computationally intractable. See [NS17] for a tractable approximation based on variational inference (Section 10.1).

3.6 Hierarchical priors

Bayesian models require specifying a prior $p(\boldsymbol{\theta})$ for the parameters. The parameters of the prior are called **hyperparameters**, and will be denoted by $\boldsymbol{\xi}$. If these are unknown, we can put a prior on them; this defines a **hierarchical Bayesian model**, or **multi-level model**, which can visualize like this: $\boldsymbol{\xi} \rightarrow \boldsymbol{\theta} \rightarrow \mathcal{D}$. We assume the prior on the hyper-parameters is fixed (e.g., we may use some kind of minimally informative prior), so the joint distribution has the form

$$p(\boldsymbol{\xi}, \boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\xi})p(\boldsymbol{\theta}|\boldsymbol{\xi})p(\mathcal{D}|\boldsymbol{\theta}) \quad (3.245)$$

The hope is that we can learn the hyperparameters by treating the parameters themselves as datapoints.

A common setting in which such an approach makes sense is when we have $J > 1$ related datasets, \mathcal{D}_j , each with their own parameters $\boldsymbol{\theta}_j$. Inferring $p(\boldsymbol{\theta}_j|\mathcal{D}_j)$ independently for each group j can give poor results if \mathcal{D}_j is a small dataset (e.g., if condition j corresponds to a rare combination of features, or a sparsely populated region). We could of course pool all the data to compute a single model, $p(\boldsymbol{\theta}|\mathcal{D})$, but that would not let us model the subpopulations. A hierarchical Bayesian model lets us

borrow statistical strength from groups with lots of data (and hence well-informed posteriors $p(\boldsymbol{\theta}_j|\mathcal{D})$) in order to help groups with little data (and hence highly uncertain posteriors $p(\boldsymbol{\theta}_j|\mathcal{D})$). The idea is that well-informed groups j will have a good estimate of $\boldsymbol{\theta}_j$, from which we can infer $\boldsymbol{\xi}$, which can be used to help estimate $\boldsymbol{\theta}_k$ for groups k with less data. (Information is shared via the hidden common parent node $\boldsymbol{\xi}$ in the graphical model, as shown in Figure 3.14.) We give some examples of this below.

After fitting such models, we can compute two kinds of posterior predictive distributions. If we want to predict observations for an existing group j , we need to use

$$p(y_j|\mathcal{D}) = \int p(y_j|\boldsymbol{\theta}_j)p(\boldsymbol{\theta}_j|\mathcal{D})d\boldsymbol{\theta}_j \quad (3.246)$$

However, if we want to predict observations for a new group $*$ that has not yet been measured, but which is comparable to (or **exchangeable with**) the existing groups $1 : J$, we need to use

$$p(y_*|\mathcal{D}) = \int p(y_*|\boldsymbol{\theta}_*)p(\boldsymbol{\theta}_*|\boldsymbol{\xi})p(\boldsymbol{\xi}|\mathcal{D})d\boldsymbol{\theta}_*d\boldsymbol{\xi} \quad (3.247)$$

We give some examples below. (More information can be found in e.g., [GH07; Gel+14a].)

3.6.1 A hierarchical binomial model

Suppose we want to estimate the prevalence of some disease amongst different group of individuals, either people or animals. Let N_j be the size of the j 'th group, and let y_j be the number of positive cases for group $j = 1 : J$. We assume $y_j \sim \text{Bin}(N_j, \theta_j)$, and we want to estimate the rates θ_j . Since some groups may have small population sizes, we may get unreliable results if we estimate each θ_j separately; for example we may observe $y_j = 0$ resulting in $\hat{\theta}_j = 0$, even though the true infection rate is higher.

One solution is to assume all the θ_j are the same; this is called **parameter tying**. The resulting pooled MLE is just $\hat{\theta}_{\text{pooled}} = \frac{\sum_j y_j}{\sum_j N_j}$. But the assumption that all the groups have the same rate is a rather strong one. A compromise approach is to assume that the θ_j are similar, but that there may be group-specific variations. This can be modeled by assuming the θ_j are drawn from some common distribution, say $\theta_j \sim \text{Beta}(a, b)$. The full joint distribution can be written as

$$p(\mathcal{D}, \boldsymbol{\theta}, \boldsymbol{\xi}) = p(\boldsymbol{\xi})p(\boldsymbol{\theta}|\boldsymbol{\xi})p(\mathcal{D}|\boldsymbol{\theta}) = p(\boldsymbol{\xi}) \left[\prod_{j=1}^J \text{Beta}(\theta_j|\boldsymbol{\xi}) \right] \left[\prod_{j=1}^J \text{Bin}(y_j|N_j, \theta_j) \right] \quad (3.248)$$

where $\boldsymbol{\xi} = (a, b)$. In Figure 3.14 we represent these assumptions using a directed graphical model (see Section 4.2.8 for an explanation of such diagrams).

It remains to specify the prior $p(\boldsymbol{\xi})$. Following [Gel+14a, p110], we use

$$p(a, b) \propto (a + b)^{-5/2} \quad (3.249)$$

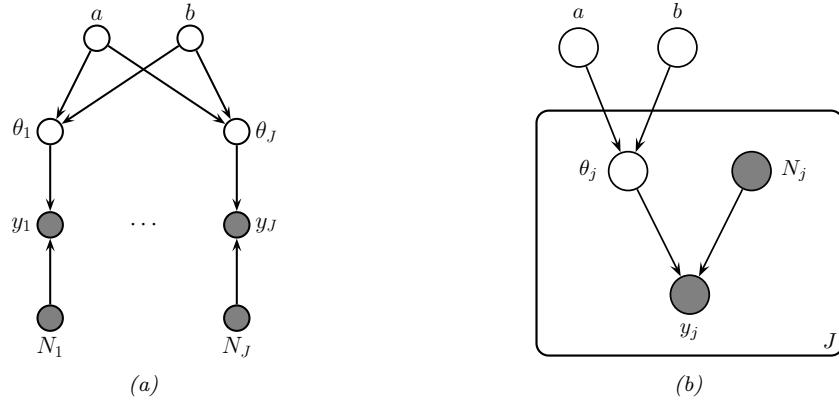


Figure 3.14: PGM for a hierarchical binomial model. (a) “Unrolled” model. (b) Same model, using plate notation.

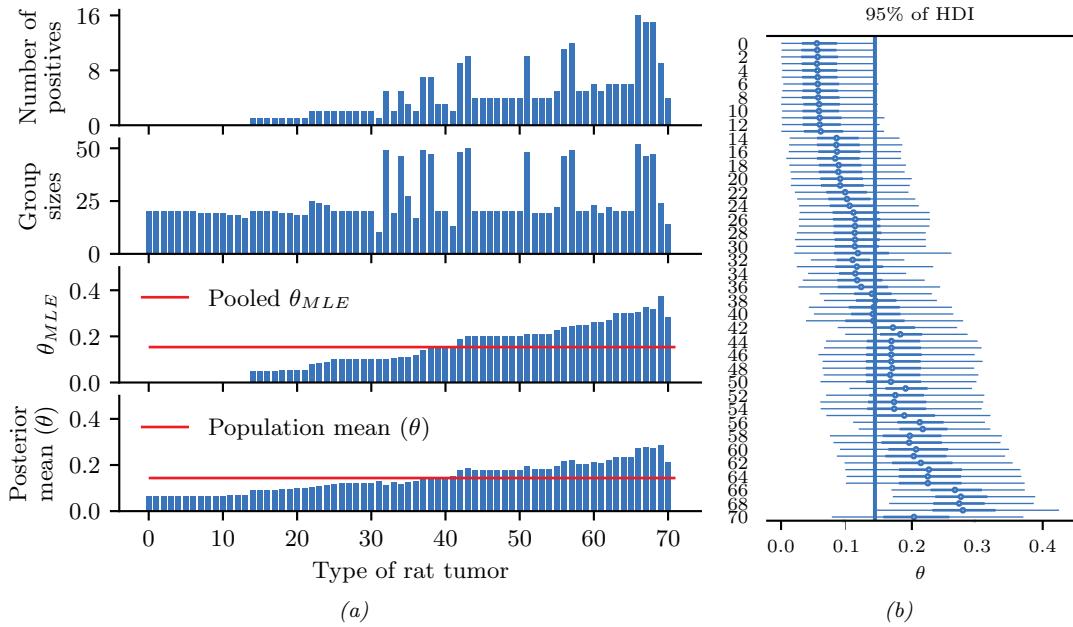


Figure 3.15: Data and inferences for the hierarchical binomial model fit using HMC. Generated by [hierarchical_binom_rats.ipynb](#).

3.6.1.1 Posterior inference

We can perform approximate posterior inference in this model using a variety of methods. In Section 3.7.1 we discuss an optimization based approach, but here we discuss one of the most popular methods in Bayesian statistics, known as HMC or Hamiltonian Monte Carlo. This is described in Section 12.5, but in short it is a form of MCMC (Markov chain Monte Carlo) that exploits information from the gradient of the log joint to guide the sampling process. This algorithm generates samples in an unconstrained parameter space, so we need to define the log joint over all the parameters $\omega = (\tilde{\theta}, \tilde{\xi}) \in \mathbb{R}^D$ as follows:

$$\log p(\mathcal{D}, \omega) = \log p(\mathcal{D}|\theta) + \log p(\theta|\xi) + \log p(\xi) \quad (3.250)$$

$$+ \sum_{j=1}^J \log |\text{Jac}(\sigma)(\tilde{\theta}_j)| + \sum_{i=1}^2 \log |\text{Jac}(\sigma_+)(\tilde{\xi}_i)| \quad (3.251)$$

where $\theta_j = \sigma(\tilde{\theta}_j)$ is the sigmoid transform, and $\xi_i = \sigma_+(\tilde{\xi}_i)$ is the softplus transform. (We need to add the Jacobian terms to account for these deterministic transformations.) We can then use automatic differentiation to compute $\nabla_\omega \log p(\mathcal{D}, \omega)$, which we pass to the HMC algorithm. This algorithm returns a set of (correlated) samples from the posterior, $(\tilde{\xi}^s, \tilde{\theta}^s) \sim p(\omega|\mathcal{D})$, which we can back transform to (ξ^s, θ^s) . We can then estimate the posterior over quantities of interest by using a Monte Carlo approximation to $p(f(\theta)|\mathcal{D})$ for suitable f (e.g., to compute the posterior mean rate for group j , we set $f(\theta) = \theta_j$).

3.6.1.2 Example: the rats dataset

In this section, we apply this model to analyze the number of rats that develop a certain kind of tumor during a particular clinical trial (see [Gel+14a, p102] for details). We show the raw data in rows 1–2 of Figure 3.15a. In row 3, we show the MLE $\hat{\theta}_j$ for each group. We see that some groups have $\hat{\theta}_j = 0$, which is much less than the pooled MLE $\hat{\theta}_{\text{pooled}}$ (red line). In row 4, we show the posterior mean $\mathbb{E}[\theta_j|\mathcal{D}]$ estimated from all the data, as well as the population mean $\mathbb{E}[\theta|\mathcal{D}] = \mathbb{E}[a/(a+b)|\mathcal{D}]$ shown in the red lines. We see that groups that have low counts have their estimates increased towards the population mean, and groups that have large counts have their estimates decreased towards the population mean. In other words, the groups regularize each other; this phenomenon is called **shrinkage**. The amount of shrinkage is controlled by the prior on (a, b) , which is inferred from the data.

In Figure 3.15b, we show the 95% credible intervals for each parameter, as well as the overall population mean. (This is known as a **forest plot**.) We can use this to decide if any group is significantly different than any specified target value (e.g., the overall average).

3.6.2 A hierarchical Gaussian model

In this section, we consider a variation of the model in Section 3.6.1, where this time we have real-valued data instead of binary count data. More specifically we assume $y_{ij} \sim \mathcal{N}(\theta_j, \sigma^2)$, where θ_j is the unknown mean for group j , and σ^2 is the observation variance (assumed to be shared across groups and fixed, for simplicity). Note that having N_j observations y_{ij} each with variance σ^2 is like having one measurement $y_j \triangleq \frac{1}{N_j} \sum_{i=1}^{N_j} y_{ij}$ with variance $\sigma_j^2 \triangleq \sigma^2/N_j$. This lets us simplify notation

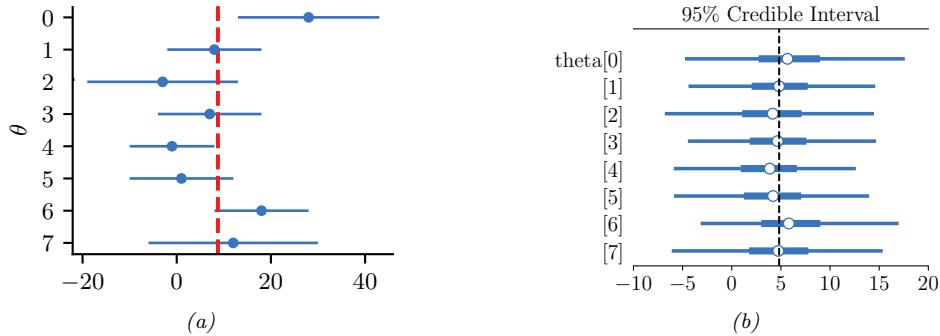


Figure 3.16: Eight schools dataset. (a) Raw data. Each row plots $y_j \pm \sigma_j$. Vertical line is the pooled estimate. (b) Posterior 95% credible intervals for θ_j . Vertical line is posterior mean $\mathbb{E}[\mu|\mathcal{D}]$. Generated by [schools8.ipynb](#).

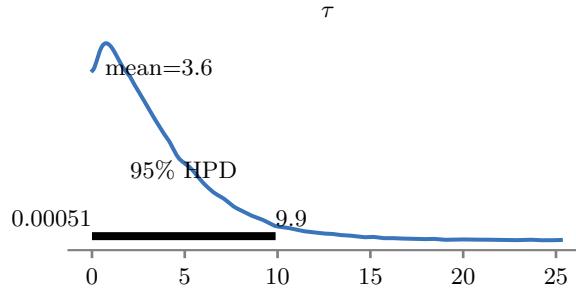


Figure 3.17: Marginal posterior density $p(\tau|\mathcal{D})$ for the 8-schools dataset. Generated by [schools8.ipynb](#).

and use one observation per group, with likelihood $y_j \sim \mathcal{N}(\theta_j, \sigma_j^2)$, where we assume the σ_j 's are known.

We use a hierarchical model by assuming each group's parameters come from a common distribution, $\theta_j \sim \mathcal{N}(\mu, \tau^2)$. The model becomes

$$p(\mu, \tau^2, \boldsymbol{\theta}_{1:J} | \mathcal{D}) \propto p(\mu)p(\tau^2) \prod_{j=1}^J \mathcal{N}(\theta_j | \mu, \tau^2) \mathcal{N}(y_j | \theta_j, \sigma_j^2) \quad (3.252)$$

where $p(\mu)p(\tau^2)$ is some kind of prior over the hyper-parameters. See Figure 3.19a for the graphical model.

3.6.2.1 Example: the eight schools dataset

Let us now apply this model to some data. We will consider the **eight schools** dataset from [Gel+14a, Sec 5.5]. The goal is to estimate the effects on a new coaching program on SAT scores. Let y_{nj} be the observed improvement in score for student n in school j compared to a baseline. Since each school has multiple students, we summarize its data using the empirical mean $\bar{y}_{\cdot j} = \frac{1}{N_j} \sum_{n=1}^{N_j} y_{nj}$

and standard deviation σ_j . See Figure 3.16a for an illustration of the data. We also show the pooled MLE for θ , which is a precision weighted average of the data:

$$\bar{y}_{..} = \frac{\sum_{j=1}^J \frac{1}{\sigma_j^2} \bar{y}_{.j}}{\sum_{j=1}^J \frac{1}{\sigma_j^2}} \quad (3.253)$$

We see that school 0 has an unusually large improvement (28 points) compared to the overall mean, suggesting that the estimating θ_0 just based on \mathcal{D}_0 might be unreliable. However, we can easily apply our hierarchical model. We will use HMC to do approximate inference. (See Section 3.7.2 for a faster approximate method.)

After computing the (approximate) posterior, we can compute the marginal posteriors $p(\theta_j|\mathcal{D})$ for each school. These distributions are shown in Figure 3.16b. Once again, we see shrinkage towards the global mean $\bar{\mu} = \mathbb{E}[\mu|\mathcal{D}]$, which is close to the pooled estimate $\bar{y}_{..}$. In fact, if we fix the hyper-parameters to their posterior mean values, and use the approximation

$$p(\mu, \tau^2|\mathcal{D}) = \delta(\mu - \bar{\mu})\delta(\tau^2 - \bar{\tau}^2) \quad (3.254)$$

then we can use the results from Section 3.4.3.1 to compute the marginal posteriors

$$p(\theta_j|\mathcal{D}) \approx p(\theta_j|\mathcal{D}_j, \bar{\mu}, \bar{\tau}^2) \quad (3.255)$$

In particular, we can show that the posterior mean $\mathbb{E}[\theta_j|\mathcal{D}]$ is in between the MLE $\hat{\theta}_j = y_j$ and the global mean $\bar{\mu} = \mathbb{E}[\mu|\mathcal{D}]$:

$$\mathbb{E}[\theta_j|\mathcal{D}, \bar{\mu}, \bar{\tau}^2] = w_j \bar{\mu} + (1 - w_j)\hat{\theta}_j \quad (3.256)$$

where the amount of shrinkage towards the global mean is given by

$$w_j = \frac{\sigma_j^2}{\sigma_j^2 + \tau^2} \quad (3.257)$$

Thus we see that there is more shrinkage for groups with smaller measurement precision (e.g., due to smaller sample size), which makes intuitive sense. There is also more shrinkage if τ^2 is smaller; of course τ^2 is unknown, but we can compute a posterior for it, as shown in Figure 3.17.

3.6.2.2 Non-centered parameterization

It turns out that posterior inference in this model is difficult for many algorithms because of the tight dependence between the variance hyperparameter τ^2 and the group means θ_j , as illustrated by the **funnel shape** in Figure 3.18. In particular, consider making local moves through parameter space. The algorithm can only “visit” the place where τ^2 is small (corresponding to strong shrinkage to the prior) if all the θ_j are close to the prior mean μ . It may be hard to move into the area where τ^2 is small unless all groups *simultaneously* move their θ_j estimates closer to μ .

A standard solution to this problem is to rewrite the model using the following **non-centered parameterization**:

$$\theta_j = \mu + \tau \eta_j \quad (3.258)$$

$$\eta_j \sim \mathcal{N}(0, 1) \quad (3.259)$$

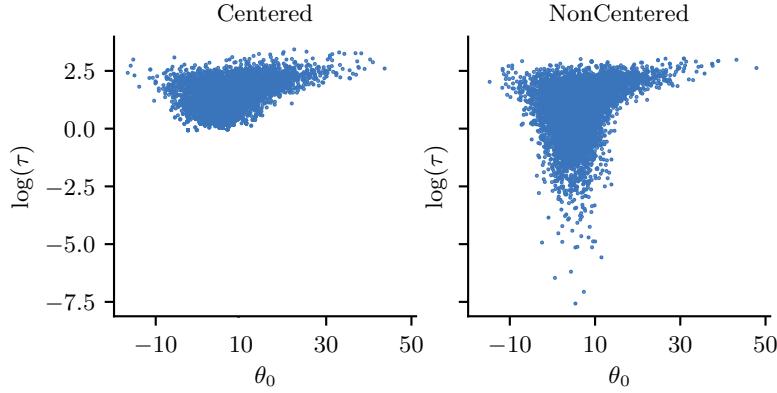


Figure 3.18: Posterior $p(\theta_0, \log(\tau) | \mathcal{D})$ for the eight schools model using (a) centered parameterization and (b) non-centered parameterization. Generated by [schools8.ipynb](#).

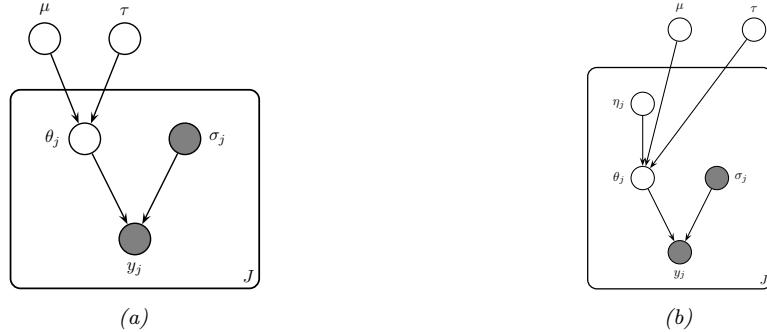


Figure 3.19: A hierarchical Gaussian Bayesian model. (a) Centered parameterization. (b) Non-centered parameterization.

See Figure 3.19b for the corresponding graphical model. By writing θ_j as a deterministic function of its parents plus a local noise term, we have reduced the dependence between θ_j and τ and hence the other θ_k variables, which can improve the computational efficiency of inference algorithms, as we discuss in Section 12.6.5. This kind of reparameterization is widely used in hierarchical Bayesian models.

3.6.3 Hierarchical conditional models

In Section 15.5, we discuss hierarchical Bayesian GLM models, which learn conditional distributions $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_j)$ for each group j , using a prior of the form $p(\boldsymbol{\theta}_j|\boldsymbol{\xi})$. In Section 17.6, we discuss hierarchical Bayesian neural networks, which generalize this idea to nonlinear predictors.

3.7 Empirical Bayes

In Section 3.6, we discussed hierarchical Bayes as a way to infer parameters from data. Unfortunately, posterior inference in such models can be computationally challenging. In this section, we discuss a computationally convenient approximation, in which we first compute a point estimate of the hyperparameters, $\hat{\xi}$, and then compute the conditional posterior, $p(\theta|\hat{\xi}, \mathcal{D})$, rather than the joint posterior, $p(\theta, \xi|\mathcal{D})$.

To estimate the hyper-parameters, we can maximize the marginal likelihood:

$$\hat{\xi}_{\text{mml}}(\mathcal{D}) = \underset{\xi}{\operatorname{argmax}} p(\mathcal{D}|\xi) = \underset{\xi}{\operatorname{argmax}} \int p(\mathcal{D}|\theta)p(\theta|\xi)d\theta \quad (3.260)$$

This technique is known as **type II maximum likelihood**, since we are optimizing the hyper-parameters, rather than the parameters. (In the context of neural networks, this is sometimes called the **evidence procedure** [Mac92a; WS93; Mac99].) Once we have estimated $\hat{\xi}$, we compute the posterior $p(\theta|\hat{\xi}, \mathcal{D})$ in the usual way.

Since we are estimating the prior parameters from data, this approach is **empirical Bayes (EB)** [CL96]. This violates the principle that the prior should be chosen independently of the data. However, we can view it as a computationally cheap approximation to inference in the full hierarchical Bayesian model, just as we viewed MAP estimation as an approximation to inference in the one level model $\theta \rightarrow \mathcal{D}$. In fact, we can construct a hierarchy in which the more integrals one performs, the “more Bayesian” one becomes, as shown below.

Method	Definition
Maximum likelihood	$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} p(\mathcal{D} \theta)$
MAP estimation	$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} p(\mathcal{D} \theta)p(\theta \xi)$
ML-II (empirical Bayes)	$\hat{\xi} = \underset{\xi}{\operatorname{argmax}} \int p(\mathcal{D} \theta)p(\theta \xi)d\theta$
MAP-II	$\hat{\xi} = \underset{\xi}{\operatorname{argmax}} \int p(\mathcal{D} \theta)p(\theta \xi)p(\xi)d\theta$
Full Bayes	$p(\theta, \xi \mathcal{D}) \propto p(\mathcal{D} \theta)p(\theta \xi)p(\xi)$

Note that ML-II is less likely to overfit than “regular” maximum likelihood, because there are typically fewer hyper-parameters ξ than there are parameters θ . We give some simple examples below, and will see more applications later in the book.

3.7.1 EB for the hierarchical binomial model

In this section, we revisit the hierarchical binomial model from Section 3.6.1, but we use empirical Bayes instead of full Bayesian inference. We can analytically integrate out the θ_j ’s, and write down

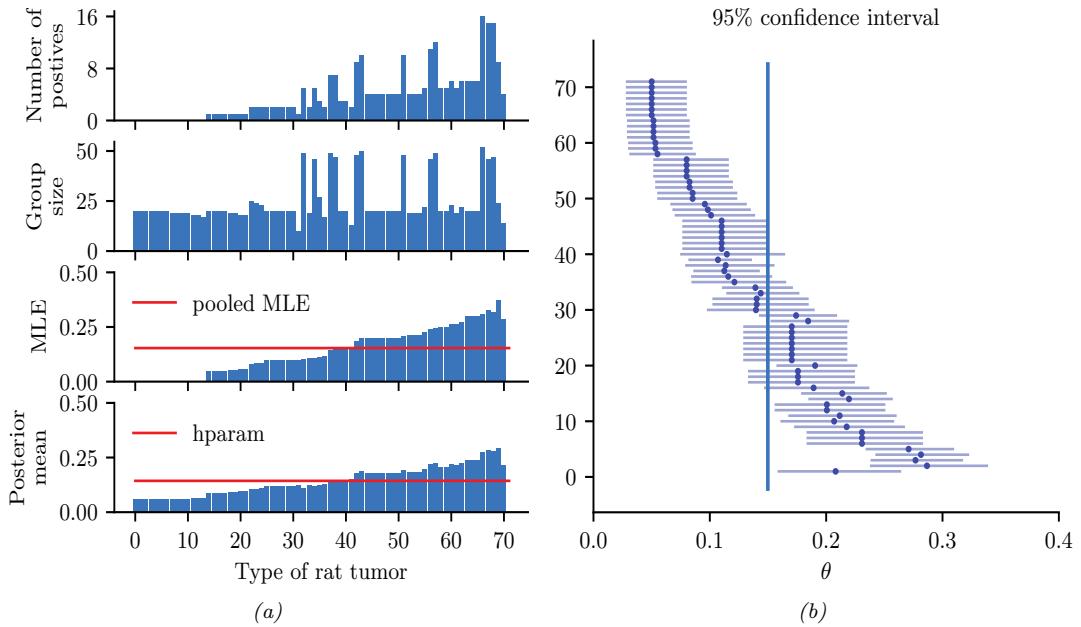


Figure 3.20: Data and inferences for the hierarchical binomial model fit using empirical Bayes. Generated by [eb_binom.ipynb](#).

the marginal likelihood directly: The resulting expression is

$$p(\mathcal{D}|\xi) = \prod_j \int \text{Bin}(y_j|N_j, \theta_j) \text{Beta}(\theta_j|a, b) d\theta_j \quad (3.261)$$

$$\propto \prod_j \frac{B(a + y_j, b + N_j - y_j)}{B(a, b)} \quad (3.262)$$

$$= \prod_j \frac{\Gamma(a + b)}{\Gamma(a)\Gamma(b)} \frac{\Gamma(a + y_j)\Gamma(b + N_j - y_j)}{\Gamma(a + b + N_j)} \quad (3.263)$$

Various ways of maximizing this marginal likelihood wrt a and b are discussed in [Min00c].

Having estimated the hyper-parameters a and b , we can plug them in to compute the posterior $p(\theta_j|\hat{a}, \hat{b}, \mathcal{D})$ for each group, using conjugate analysis in the usual way. We show the results in Figure 3.20; they are very similar to the full Bayesian analysis shown in Figure 3.15, but the EB method is much faster.

3.7.2 EB for the hierarchical Gaussian model

In this section, we revisit the hierarchical Gaussian model from Section 3.6.2.1. However, we fit the model using empirical Bayes.

For simplicity, we will assume that $\sigma_j^2 = \sigma^2$ is the same for all groups. When the variances are equal, we can derive the EB estimate in closed form, as we now show. We have

$$p(y_j|\mu, \tau^2, \sigma^2) = \int \mathcal{N}(y_j|\theta_j, \sigma^2) \mathcal{N}(\theta_j|\mu, \tau^2) d\theta_j = \mathcal{N}(y_j|\mu, \tau^2 + \sigma^2) \quad (3.264)$$

Hence the marginal likelihood is

$$p(\mathcal{D}|\mu, \tau^2, \sigma^2) = \prod_{j=1}^J \mathcal{N}(y_j|\mu, \tau^2 + \sigma^2) \quad (3.265)$$

Thus we can estimate the hyper-parameters using the usual MLEs for a Gaussian. For μ , we have

$$\hat{\mu} = \frac{1}{J} \sum_{j=1}^J y_j = \bar{y} \quad (3.266)$$

which is the overall mean. For τ^2 , we can use moment matching, which is equivalent to the MLE for a Gaussian. This means we equate the model variance to the empirical variance:

$$\hat{\tau}^2 + \sigma^2 = \frac{1}{J} \sum_{j=1}^J (y_j - \bar{y})^2 \triangleq v \quad (3.267)$$

so $\hat{\tau}^2 = v - \sigma^2$. Since we know τ^2 must be positive, it is common to use the following revised estimate:

$$\hat{\tau}^2 = \max\{0, v - \sigma^2\} = (v - \sigma^2)_+ \quad (3.268)$$

Given this, the posterior mean becomes

$$\hat{\theta}_j = \lambda\mu + (1 - \lambda)y_j = \mu + (1 - \lambda)(y_j - \mu) \quad (3.269)$$

where $\lambda_j = \lambda = \sigma^2 / (\sigma^2 + \tau^2)$.

Unfortunately, we cannot use the above method on the 8-schools dataset in Section 3.6.2.1, since it uses unequal σ_j . However, we can still use the EM algorithm or other optimization based methods.

3.7.3 EB for Markov models (n-gram smoothing)

The main problem with add-one smoothing, discussed in Section 2.6.3.3, is that it assumes that all n-grams are equally likely, which is not very realistic. A more sophisticated approach, called **deleted interpolation** [CG96], defines the transition matrix as a convex combination of the bigram frequencies $f_{jk} = N_{jk}/N_j$ and the unigram frequencies $f_k = N_k/N$:

$$A_{jk} = (1 - \lambda)f_{jk} + \lambda f_k = (1 - \lambda)\frac{N_{jk}}{N_j} + \lambda\frac{N_k}{N} \quad (3.270)$$

The term λ is usually set by cross validation. There is also a closely related technique called **backoff smoothing**; the idea is that if f_{jk} is too small, we “back off” to a more reliable estimate, namely f_k .

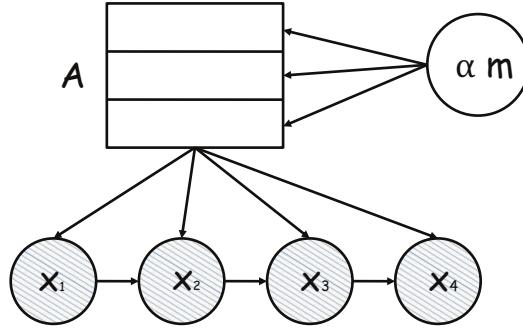


Figure 3.21: A Markov chain in which we put a different Dirichlet prior on every row of the transition matrix \mathbf{A} , but the hyperparameters of the Dirichlet are shared.

We now show that this heuristic can be interpreted as an empirical Bayes approximation to a hierarchical Bayesian model for the parameter vectors corresponding to each row of the transition matrix \mathbf{A} . Our presentation follows [MP95].

First, let us use an independent Dirichlet prior on each row of the transition matrix:

$$\mathbf{A}_j \sim \text{Dir}(\alpha_0 m_1, \dots, \alpha_0 m_K) = \text{Dir}(\alpha_0 \mathbf{m}) = \text{Dir}(\boldsymbol{\alpha}) \quad (3.271)$$

where \mathbf{A}_j is row j of the transition matrix, \mathbf{m} is the prior mean (satisfying $\sum_k m_k = 1$) and α_0 is the prior strength (see Figure 3.21). In terms of the earlier notation, we have $\boldsymbol{\theta}_j = \mathbf{A}_j$ and $\boldsymbol{\xi} = (\boldsymbol{\alpha}, \mathbf{m})$.

The posterior is given by $\mathbf{A}_j \sim \text{Dir}(\boldsymbol{\alpha} + \mathbf{N}_j)$, where $\mathbf{N}_j = (N_{j1}, \dots, N_{jK})$ is the vector that records the number of times we have transitioned out of state j to each of the other states. The posterior predictive density is

$$p(X_{t+1} = k | X_t = j, \mathcal{D}) = \frac{N_{jk} + \alpha_j m_k}{N_j + \alpha_0} = \frac{f_{jk} N_j + \alpha_j m_k}{N_j + \alpha_0} \quad (3.272)$$

$$= (1 - \lambda_j) f_{jk} + \lambda_j m_k \quad (3.273)$$

where

$$\lambda_j = \frac{\alpha_j}{N_j + \alpha_0} \quad (3.274)$$

This is very similar to Equation (3.270) but not identical. The main difference is that the Bayesian model uses a context-dependent weight λ_j to combine m_k with the empirical frequency f_{jk} , rather than a fixed weight λ . This is like *adaptive* deleted interpolation. Furthermore, rather than backing off to the empirical marginal frequencies f_k , we back off to the model parameter m_k .

The only remaining question is: what values should we use for $\boldsymbol{\alpha}$ and \mathbf{m} ? Let's use empirical Bayes. Since we assume each row of the transition matrix is a priori independent given $\boldsymbol{\alpha}$, the marginal likelihood for our Markov model is given by

$$p(\mathcal{D} | \boldsymbol{\alpha}) = \prod_j \frac{B(\mathbf{N}_j + \boldsymbol{\alpha})}{B(\boldsymbol{\alpha})} \quad (3.275)$$

where $\mathbf{N}_j = (N_{j1}, \dots, N_{jK})$ are the counts for leaving state j and $B(\boldsymbol{\alpha})$ is the generalized beta function.

We can fit this using the methods discussed in [Min00c]. However, we can also use the following approximation [MP95, p12]:

$$m_k \propto |\{j : N_{jk} > 0\}| \quad (3.276)$$

This says that the prior probability of word k is given by the number of different contexts in which it occurs, rather than the number of times it occurs. To justify the reasonableness of this result, MacKay and Peto [MP95] give the following example.

Imagine, you see, that the language, you see, has, you see, a frequently occurring couplet 'you see', you see, in which the second word of the couplet, see, follows the first word, you, with very high probability, you see. Then the marginal statistics, you see, are going to become hugely dominated, you see, by the words you and see, with equal frequency, you see.

If we use the standard smoothing formula, Equation (3.270), then $P(\text{you}|\text{novel})$ and $P(\text{see}|\text{novel})$, for some novel context word not seen before, would turn out to be the same, since the marginal frequencies of 'you' and 'see' are the same (11 times each). However, this seems unreasonable. 'You' appears in many contexts, so $P(\text{you}|\text{novel})$ should be high, but 'see' only follows 'you', so $P(\text{see}|\text{novel})$ should be low. If we use the Bayesian formula Equation (3.273), we will get this effect for free, since we back off to m_k not f_k , and m_k will be large for 'you' and small for 'see' by Equation (3.276).

Although elegant, this Bayesian model does not beat the state-of-the-art language model, known as **interpolated Kneser-Ney** [KN95; CG98]. By using ideas from nonparametric Bayes, one can create a language model that outperforms such heuristics, as discussed in [Teh06; Woo+09]. However, one can get even better results using recurrent neural nets (Section 16.3.4); the key to their success is that they don't treat each symbol "atomically", but instead learn a distributed embedding representation, which encodes the assumption that some symbols are more similar to each other than others.

3.7.4 EB for non-conjugate models

For more complex models, we cannot compute the EB estimate exactly. However, we can use the variational EM method to compute an approximate EB estimate, as we explain in Section 10.1.3.2.

3.8 Model selection

All models are wrong, but some are useful. — George Box [BD87, p424].⁹

In this section, we assume we have a set of different models \mathcal{M} , each of which may fit the data to different degrees, and each of which may make different assumptions. We discuss how to pick the best model from this set, or to average over all of them.

We assume the "true" model is in the set \mathcal{M} ; this is known as the **\mathcal{M} -complete** assumption [BS94]. Of course, in reality, none of the models may be adequate; this is known as the **\mathcal{M} -open** scenario

9. George Box is a retired statistics professor at the University of Wisconsin.

[BS94; CI13]. We can check how well a model fits (or fails to fit) the data using the procedures in Section 3.9. If none of the models are a good fit, we need to expand our hypothesis space.

3.8.1 Bayesian model selection

The natural way to pick the best model is to pick the most probable model according to Bayes' rule:

$$\hat{m} = \operatorname{argmax}_{m \in \mathcal{M}} p(m|\mathcal{D}) \quad (3.277)$$

where

$$p(m|\mathcal{D}) = \frac{p(\mathcal{D}|m)p(m)}{\sum_{m \in \mathcal{M}} p(\mathcal{D}|m)p(m)} \quad (3.278)$$

is the posterior over models. This is called **Bayesian model selection**. If the prior over models is uniform, $p(m) = 1/|\mathcal{M}|$, then the MAP model is given by

$$\hat{m} = \operatorname{argmax}_{m \in \mathcal{M}} p(\mathcal{D}|m) \quad (3.279)$$

The quantity $p(\mathcal{D}|m)$ is given by

$$p(\mathcal{D}|m) = \int p(\mathcal{D}|\boldsymbol{\theta}, m)p(\boldsymbol{\theta}|m)d\boldsymbol{\theta} \quad (3.280)$$

This is known as the **marginal likelihood**, or the **evidence** for model m . (See Section 3.8.3 for details on how to compute this quantity.) If the model assigns high prior predictive density to the observed data, then we deem it a good model. If, however, the model has too much flexibility, then some prior settings will not match the data; this probability mass will be “wasted”, lowering the expected likelihood. This implicit regularization effect is called the **Bayesian Occam’s razor**. See Figure 3.22 for an illustration.

3.8.1.1 Example: is the coin fair?

As an example, suppose we observe some coin tosses, and want to decide if the data was generated by a fair coin, $\theta = 0.5$, or a potentially biased coin, where θ could be any value in $[0, 1]$. Let us denote the first model by M_0 and the second model by M_1 . The marginal likelihood under M_0 is simply

$$p(\mathcal{D}|M_0) = \left(\frac{1}{2}\right)^N \quad (3.281)$$

where N is the number of coin tosses. From Equation (3.31), the marginal likelihood under M_1 , using a Beta prior, is

$$p(\mathcal{D}|M_1) = \int p(\mathcal{D}|\theta)p(\theta)d\theta = \frac{B(\alpha_1 + N_1, \alpha_0 + N_0)}{B(\alpha_1, \alpha_0)} \quad (3.282)$$

We plot $\log p(\mathcal{D}|M_1)$ vs the number of heads N_1 in Figure 3.23(a), assuming $N = 5$ and a uniform prior, $\alpha_1 = \alpha_0 = 1$. (The shape of the curve is not very sensitive to α_1 and α_0 , as long as the

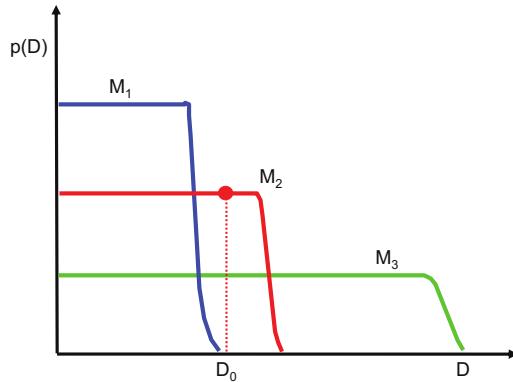


Figure 3.22: A schematic illustration of the Bayesian Occam’s razor. The broad (green) curve corresponds to a complex model, the narrow (blue) curve to a simple model, and the middle (red) curve is just right. Adapted from Figure 3.13 of [Bis06]. See also [MG05, Figure 2] for a similar plot produced on real data.

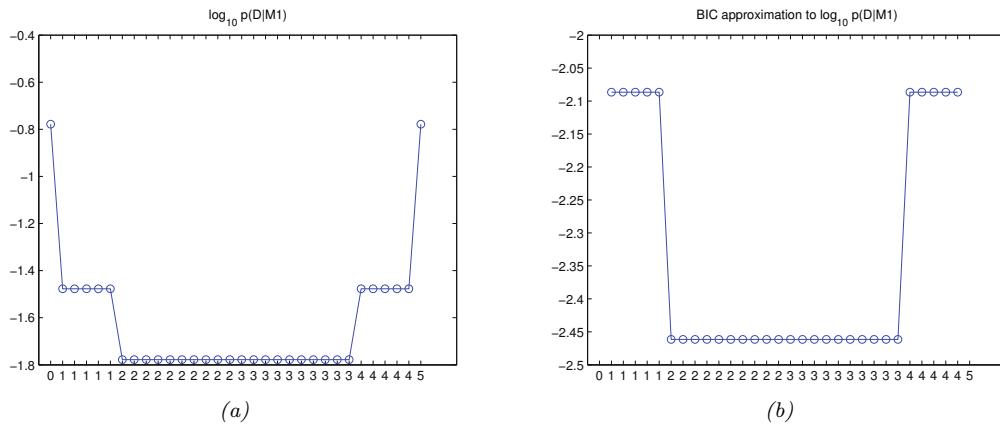


Figure 3.23: (a) Log marginal likelihood vs number of heads for the coin tossing example. (b) BIC approximation. (The vertical scale is arbitrary, since we are holding N fixed.) Generated by `coins_model_sel_demo.ipynb`.

prior is symmetric, so $\alpha_0 = \alpha_1$.) If we observe 2 or 3 heads, the unbiased coin hypothesis M_0 is more likely than M_1 , since M_0 is a simpler model (it has no free parameters) — it would be a suspicious coincidence if the coin were biased but happened to produce almost exactly 50/50 heads/tails. However, as the counts become more extreme, we favor the biased coin hypothesis.

In Figure 3.23(b), we show a similar result, where we approximate the log marginal likelihood with the BIC score (see Section 3.8.7.2).

3.8.2 Bayes model averaging

If our goal is to perform predictions, we are better off averaging over all models, rather than predicting using just one single model. That is, we should compute the **posterior predictive distribution** using

$$p(y|\mathcal{D}) = \sum_{m \in \mathcal{M}} p(y|m)p(m|\mathcal{D}) \quad (3.283)$$

This is called **Bayes model averaging** [Hoe+99]. This is similar to the machine learning technique of **ensembling**, in which we take a weighted combination of predictors. However, it is not the same, as pointed out in [Min00b], since the weights in an ensemble do not need to sum to 1. In particular, in BMA, if there is a single best model, call it m^* , then in the large sample limit, $p(m|\mathcal{D})$ will become a degenerate distribution with all its weight on m^* , and the other members of \mathcal{M} will be ignored. This does not happen with an ensemble.

3.8.3 Estimating the marginal likelihood

To perform Bayesian model selection or averaging, we need to be able to compute the marginal likelihood in Equation (3.280), also called the evidence. Below we give a brief summary of some suitable methods. For more details, see e.g., [FW12].

3.8.3.1 Analytic solution for conjugate models

If we use a conjugate prior, we can compute the marginal likelihood analytically, as we discussed in Section 3.4.5.4. We give a worked example in Section 3.8.1.1.

3.8.3.2 Harmonic mean estimator

A particularly simple estimator, known as the **harmonic mean estimator**, was proposed in [NR94]. It is defined as follows:

$$p(\mathcal{D}) \approx \left(\frac{1}{S} \sum_{s=1}^S \frac{1}{p(\mathcal{D}|\boldsymbol{\theta}_s)} \right)^{-1} \quad (3.284)$$

where $\boldsymbol{\theta}_s \sim p(\boldsymbol{\theta})$ are samples from the prior. This follows from the following identity:

$$\mathbb{E} \left[\frac{1}{p(\mathcal{D}|\boldsymbol{\theta})} \right] = \int \frac{1}{p(\mathcal{D}|\boldsymbol{\theta})} p(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta} \quad (3.285)$$

$$= \int \frac{1}{p(\mathcal{D}|\boldsymbol{\theta})} \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} d\boldsymbol{\theta} \quad (3.286)$$

$$= \frac{1}{p(\mathcal{D})} \int p(\boldsymbol{\theta}) d\boldsymbol{\theta} = \frac{1}{p(\mathcal{D})} \quad (3.287)$$

(We have assumed the prior is proper, so it integrates to 1.)

Unfortunately, the number of samples needed to get a good estimate is generally very large, since most samples from the prior will have low likelihood, making this approach useless in practice. Indeed,

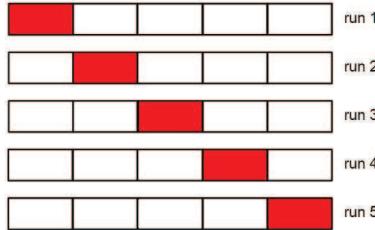


Figure 3.24: Schematic of 5-fold cross validation.

Radford Neal made a blog post in which he described this method as “The Worst Monte Carlo Method Ever”.¹⁰⁾

3.8.3.3 Other Monte Carlo methods

The marginal likelihood can be more reliably estimated using annealed importance sampling, as discussed in Section 11.5.4.1. An extension of this, known as sequential Monte Carlo sampling, as discussed in Section 13.2.3.3. Another method that is well suited to estimate the normalization constant is known as **nested sampling** [Ski06; Buc21].

3.8.3.4 Variational Bayes

An efficient way to compute an approximation to the evidence is to use variational Bayes, which we discuss in Section 10.3.3. This computes a tractable approximation to the posterior, $q(\boldsymbol{\theta}|\mathcal{D})$, by optimizing the evidence lower bound or ELBO, $\log q(\mathcal{D}|\boldsymbol{\theta})$, which can be used to approximate the evidence.

3.8.4 Connection between cross validation and marginal likelihood

A standard approach to model evaluation is to estimate its predictive performance (in terms of log likelihood) on a **validation set**, which is distinct from the training set which is used to fit the model. If we don’t have such a separate validation set, we can make one by partitioning the training set into K subsets or “**folds**”, and then training on $K - 1$ and testing on the K ’th; we repeat this K times, as shown in Figure 3.24. This is known as **cross validation**.

If we set $K = N$, the method is known as **leave-one-out cross validation** or **LOO-CV**, since we train on $N - 1$ points and test on the remaining one, and we do this N times. More precisely, we have

$$L_{\text{LOO}}(m) \triangleq \sum_{n=1}^N \log p(\mathcal{D}_n | \hat{\boldsymbol{\theta}}(\mathcal{D}_{-n}), m) \quad (3.288)$$

10. <https://bit.ly/3t7id0k>.

where $\hat{\boldsymbol{\theta}}_{-n}$ is the parameter estimate computing when we omit \mathcal{D}_n from the training set. (We discuss fast approximations to this in Section 3.8.6.)

Interestingly, the LOO-CV version of log likelihood is closely related to the log marginal likelihood. To see this, let us write the log marginal likelihood (LML) in sequential form as follows:

$$\text{LML}(m) \triangleq \log p(\mathcal{D}|m) = \log \prod_{n=1}^N p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) = \sum_{n=1}^N \log p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) \quad (3.289)$$

where

$$p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) = \int p(\mathcal{D}_n|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D}_{1:n-1}, m)d\boldsymbol{\theta} \quad (3.290)$$

Note that we evaluate the posterior on the first $n - 1$ datapoints and use this to predict the n 'th; this is called **prequential analysis** [DV99].

Suppose we use a point estimate for the parameters at time n , rather than the full posterior. We can then use a plugin approximation to the n 'th predictive distribution:

$$p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) \approx \int p(\mathcal{D}_n|\boldsymbol{\theta})\delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}_m(\mathcal{D}_{1:n-1}))d\boldsymbol{\theta} = p(\mathcal{D}_n|\hat{\boldsymbol{\theta}}_m(\mathcal{D}_{1:n-1})) \quad (3.291)$$

Then Equation (3.289) simplifies to

$$\log p(\mathcal{D}|m) \approx \sum_{n=1}^N \log p(\mathcal{D}_n|\hat{\boldsymbol{\theta}}_m(\mathcal{D}_{1:n-1}), m) \quad (3.292)$$

This is very similar to Equation (3.288), except it is evaluated sequentially. A complex model will overfit the “early” examples and will then predict the remaining ones poorly, and thus will get low marginal likelihood as well as a low cross-validation score. See [FH20] for further discussion.

3.8.5 Conditional marginal likelihood

The marginal likelihood answers the question “what is the likelihood of generating the training data from my prior?”. This can be suitable for hypothesis testing between different fixed priors, but is less useful for selecting models based on their posteriors. In the latter case, we are more interested in the question “what is the probability that the posterior could generate withheld points from the data distribution?”, which is related to the generalization performance of the (fitted) model. In fact [Lot+22] showed that the marginal likelihood can sometimes be negatively correlated with the generalization performance, because the first few terms in the LML decomposition may be large and negative for a model that has a poor prior but which otherwise adapts quickly to the data (by virtue of the prior being weak).

A better approach is to use the **conditional log marginal likelihood**, which is defined as follows [Lot+22]:

$$\text{CLML}(m) = \sum_{n=K}^N \log p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) \quad (3.293)$$

where $K \in \{1, \dots, N\}$ is a parameter of the algorithm. This evaluates the LML of the last $N - K$ datapoints, under the posterior given by the first K datapoints. We can reduce the dependence on the ordering of the datapoints by averaging over orders; if we set $K = N - 1$ and average over all orders, we get the LOO estimate.

The CLML is much more predictive of generalization performance than the LML, and is much less sensitive to prior hyperparameters. Furthermore, it is easier to calculate, since we can use a straightforward Monte Carlo estimate of the integral, where we sample from the posterior $p(\boldsymbol{\theta}|\mathcal{D}_{<n})$; this does not suffer from the same problems as the harmonic mean estimator in Section 3.8.3.2 which samples from the prior.

3.8.6 Bayesian leave-one-out (LOO) estimate

In this section we discuss a computationally efficient method, based on importance sampling, to approximate the leave-one-out (LOO) estimate without having to fit the model N times. We focus on conditional (supervised) models, so $p(\mathcal{D}|\boldsymbol{\theta}) = p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$.

Suppose we have computed the posterior given the full dataset for model m . We can use this to evaluate the resulting predictive distribution $p(\mathbf{y}_n|\mathbf{x}_n, \mathcal{D}, m)$ for each datapoint n in the dataset. This gives the **log-pointwise predictive-density** or **LPPD** score:

$$\text{LPPD}(m) \triangleq \sum_{n=1}^N \log p(\mathbf{y}_n|\mathbf{x}_n, \mathcal{D}, m) = \sum_{n=1}^N \log \int p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}, m) p(\boldsymbol{\theta}|\mathcal{D}, m) d\boldsymbol{\theta} \quad (3.294)$$

We can approximate LPPD with Monte Carlo:

$$\text{LPPD}(m) \approx \sum_{n=1}^N \log \left(\frac{1}{S} \sum_{s=1}^S p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}_s, m) \right) \quad (3.295)$$

where $\boldsymbol{\theta}_s \sim p(\boldsymbol{\theta}|\mathcal{D}, m)$ is a posterior sample.

The trouble with LPPD is that it predicts the n 'th datapoint \mathbf{y}_n using all the data, including \mathbf{y}_n . What we would like to compute is the **expected LPPD (ELPD)** on *future data*, $(\mathbf{x}_*, \mathbf{y}_*)$:

$$\text{ELPD}(m) \triangleq \mathbb{E}_{\mathbf{x}_*, \mathbf{y}_*} \log p(\mathbf{y}_*|\mathbf{x}_*, \mathcal{D}, m) \quad (3.296)$$

Of course, the future data is unknown, but we can use a LOO approximation:

$$\text{ELPD}_{\text{LOO}}(m) \triangleq \sum_{n=1}^N \log p(\mathbf{y}_n|\mathbf{x}_n, \mathcal{D}_{-n}, m) = \sum_{n=1}^N \log \int p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}, m) p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m) d\boldsymbol{\theta} \quad (3.297)$$

This is a Bayesian version of Equation (3.288). We can approximate this integral using Monte Carlo:

$$\text{ELPD}_{\text{LOO}}(m) \approx \sum_{n=1}^N \log \left(\frac{1}{S} \sum_{s=1}^S p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}_{s,-n}, m) \right) \quad (3.298)$$

where $\boldsymbol{\theta}_{s,-n} \sim p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m)$.

The above procedure requires computing N different posteriors, leaving one datapoint out at a time, which is slow. A faster alternative is to compute $p(\boldsymbol{\theta}|\mathcal{D}, m)$ once, and then use importance

sampling (Section 11.5) to approximate the above integral. More precisely, let $f(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m)$ be the target distribution of interest, and let $g(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D}, m)$ be the proposal. Define the importance weight for each sample s when leaving out example n to be

$$w_{s,-n} = \frac{f(\boldsymbol{\theta}_s)}{g(\boldsymbol{\theta}_s)} = \frac{p(\mathcal{D}_{-n}|\boldsymbol{\theta}_s)p(\boldsymbol{\theta}_s)}{p(\mathcal{D}_s|\boldsymbol{\theta}_s)} = \frac{p(\mathcal{D}_{-n}|\boldsymbol{\theta}_s)p(\boldsymbol{\theta}_s)}{p(\mathcal{D}_{-n})} \frac{p(\mathcal{D})}{p(\mathcal{D}|\boldsymbol{\theta}_s)p(\boldsymbol{\theta}_s)} \quad (3.299)$$

$$\propto \frac{p(\mathcal{D}_{-n}|\boldsymbol{\theta}_s)}{p(\mathcal{D}|\boldsymbol{\theta}_s)} = \frac{p(\mathcal{D}_{-n}|\boldsymbol{\theta}_s)}{p(\mathcal{D}_{-n}|\boldsymbol{\theta})p(\mathcal{D}_n|\boldsymbol{\theta}_s)} = \frac{1}{p(\mathcal{D}_n|\boldsymbol{\theta}_s)} \quad (3.300)$$

We then normalize the weights to get

$$\hat{w}_{s,-n} = \frac{w_{s,-n}}{\sum_{s'=1}^S w_{s',-n}} \quad (3.301)$$

and use them to get the estimate

$$\text{ELPD}_{\text{IS-LOO}}(m) = \sum_{n=1}^N \log \left(\sum_{s=1}^S \hat{w}_{s,-n} p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}_s, m) \right) \quad (3.302)$$

Unfortunately, the importance weights may have high variance, where some weights are much larger than others. To reduce this effect, we fit a Pareto distribution (Section 2.2.3.5) to each set of weights for each sample, and use this to smooth the weights. This technique is called **Pareto smoothed importance sampling** or **PSIS** [Veh+15; VGG17]. The Pareto distribution has the form

$$p(r|u, \sigma, k) = \sigma^{-1} (1 + k(r - u)\sigma^{-1})^{-1/k-1} \quad (3.303)$$

where u is the location, σ is the scale, and k is the shape. The parameter values k_n (for each datapoint n) can be used to assess how well this approximation works. If we find $k_n > 0.5$ for any given point, it is likely an outlier, and the resulting LOO estimate is likely to be quite poor. See [Siv+20] for further discussion, and [Kel21] for a general tutorial on PSIS-LOO-CV.

3.8.7 Information criteria

An alternative approach to cross validation is to score models using the negative log likelihood (or LPPD) on the training set plus a **complexity penalty** term:

$$\mathcal{L}(m) = -\log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + C(m) \quad (3.304)$$

This is called an **information criterion**. Different methods use different complexity terms $C(m)$, as we discuss below. See e.g., [GHV14] for further details.

A note on notation: it is conventional, when working with information criteria, to scale the NLL by -2 to get the **deviance**:

$$\text{deviance}(m) = -2 \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) \quad (3.305)$$

This makes the math “prettier” for certain Gaussian models.

3.8.7.1 Minimum description length (MDL)

We can think about the problem of scoring different models in terms of information theory (Chapter 5). The goal is for the sender to communicate the data to the receiver. First the sender needs to specify which model m to use; this takes $C(m) = -\log p(m)$ bits (see Section 5.2). Then the receiver can fit the model, by computing $\hat{\boldsymbol{\theta}}_m$, and can thus approximately reconstruct the data. To perfectly reconstruct the data, the sender needs to send the residual errors that cannot be explained by the model; this takes

$$-L(m) = -\log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) = -\sum_n \log p(\mathbf{y}_n|\mathbf{x}_n, \hat{\boldsymbol{\theta}}, m) \quad (3.306)$$

bits. (We are ignoring the cost of sending the input features \mathbf{x}_n , if present.) The total cost is

$$\mathcal{L}_{\text{MDL}}(m) = -\log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + C(m) \quad (3.307)$$

Choosing the model which minimizes this cost is known as the **minimum description length** or **MDL** principle. See e.g., [HY01] for details.

3.8.7.2 The Bayesian information criterion (BIC)

The **Bayesian information criterion** or **BIC** [Sch78] is similar to the MDL, and has the form

$$\mathcal{L}_{\text{BIC}}(m) = -2 \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + D_m \log N \quad (3.308)$$

where D_m is the **degrees of freedom** of model m .

We can derive the BIC score as a simple approximation to the log marginal likelihood. In particular, suppose we make a Gaussian approximation to the posterior, as discussed in Section 7.4.3. Then we get (from Equation (7.28)) the following:

$$\log p(\mathcal{D}|m) \approx \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}_{\text{map}}) + \log p(\hat{\boldsymbol{\theta}}_{\text{map}}) - \frac{1}{2} \log |\mathbf{H}| \quad (3.309)$$

where \mathbf{H} is the Hessian of the negative log joint $\log p(\mathcal{D}, \boldsymbol{\theta})$ evaluated at the MAP estimate $\hat{\boldsymbol{\theta}}_{\text{map}}$. We see that Equation (3.309) is the log likelihood plus some penalty terms. If we have a uniform prior, $p(\boldsymbol{\theta}) \propto 1$, we can drop the prior term, and replace the MAP estimate with the MLE, $\hat{\boldsymbol{\theta}}$, yielding

$$\log p(\mathcal{D}|m) \approx \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}) - \frac{1}{2} \log |\mathbf{H}| \quad (3.310)$$

We now focus on approximating the $\log |\mathbf{H}|$ term, which is sometimes called the **Occam factor**, since it is a measure of model complexity (volume of the posterior distribution). We have $\mathbf{H} = \sum_{i=1}^N \mathbf{H}_i$, where $\mathbf{H}_i = \nabla \nabla \log p(\mathcal{D}_i|\boldsymbol{\theta})$. Let us approximate each \mathbf{H}_i by a fixed matrix $\hat{\mathbf{H}}$. Then we have

$$\log |\mathbf{H}| = \log |N\hat{\mathbf{H}}| = \log(N^D|\hat{\mathbf{H}}|) = D \log N + \log |\hat{\mathbf{H}}| \quad (3.311)$$

where $D = \dim(\boldsymbol{\theta})$ and we have assumed \mathbf{H} is full rank. We can drop the $\log |\hat{\mathbf{H}}|$ term, since it is independent of N , and thus will get overwhelmed by the likelihood. Putting all the pieces together, we get the **BIC score** that we want to maximize:

$$J_{\text{BIC}}(m) = \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) - \frac{D_m}{2} \log N \quad (3.312)$$

We can also define the **BIC loss**, that we want to minimize, by multiplying by -2 :

$$\mathcal{L}_{\text{BIC}}(m) = -2 \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + D_m \log N \quad (3.313)$$

3.8.7.3 Akaike information criterion

The **Akaike information criterion** [Aka74] is closely related to BIC. It has the form

$$\mathcal{L}_{\text{AIC}}(m) = -2 \log p(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + 2D_m \quad (3.314)$$

This penalizes complex models less heavily than BIC, since the regularization term is independent of N . This estimator can be derived from a frequentist perspective.

3.8.7.4 Widely applicable information criterion (WAIC)

The main problem with MDL, BIC, and AIC is that it can be hard to compute the degrees of a freedom of a model, needed to define the complexity term, since most parameters are highly correlated and not uniquely identifiable from the likelihood. In particular, if the mapping from parameters to the likelihood is not one-to-one, then the model known as a **singular statistical model**, since the corresponding Fisher information matrix (Section 3.3.4), and hence the Hessian \mathbf{H} above, may be singular (have determinant 0). An alternative criterion that works even in the singular case is known as the **widely applicable information criterion** (WAIC), also known as the **Watanabe–Akaike information criterion** [Wat10; Wat13].

WAIC is like other information criteria, except it is more Bayesian. First it replaces the log likelihood $L(m)$, which uses a point estimate of the parameters, with the LPPD, which marginalizes them out. (see Equation (3.295)). For the complexity term, WAIC uses the variance of the predictive distribution:

$$C(m) = \sum_{n=1}^N \mathbb{V}_{\boldsymbol{\theta}|\mathcal{D},m} [\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}, m)] \approx \sum_{n=1}^N \mathbb{V}\{\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}_s, m) : s = 1 : S\} \quad (3.315)$$

The intuition for this is as follows: if, for a given datapoint n , the different posterior samples $\boldsymbol{\theta}_s$ make very different predictions, then the model is uncertain, and likely too flexible. The complexity term essentially counts how often this occurs. The final WAIC loss is

$$\mathcal{L}_{\text{WAIC}}(m) = -2\text{LPPD}(m) + 2C(m) \quad (3.316)$$

Interestingly, it can be shown that the PSIS LOO estimate in Section 3.8.6 is asymptotically equivalent to WAIC [VGG17].

3.9 Model checking

Bayesian inference and decision making is optimal, but only if the modeling assumptions are correct. In this section, we discuss some ways to assess if a model is reasonable. From a Bayesian perspective, this can seem a bit odd, since if we knew there was a better model, why don't we just use that? Here we assume that we do not have a specific alternative model in mind (so we are not performing model selection, unlike Section 3.8.1). Instead we are just trying to see if the data we observe is “typical” of what we might expect if our model were correct. This is called **model checking**.

3.9.1 Posterior predictive checks

Suppose we are trying to estimate the probability of heads for a coin, $\theta \in [0, 1]$. We have two candidate models or hypotheses, M_1 which corresponds to $\theta = 0.99$ and M_2 which corresponds to $\theta = 0.01$. Suppose we flip the coin 40 times and it comes up heads 30 times. Obviously we have $p(M = M_1 | \mathcal{D}) \gg p(M = M_2 | \mathcal{D})$. However model M_1 is still a very bad model for the data. (This example is from [Kru15, p331].)

To evaluate how good a candidate model M is, after seeing some data \mathcal{D} , we can imagine using the model to generate synthetic future datasets, by drawing from the posterior predictive distribution:

$$\tilde{\mathcal{D}}^s \sim p(\tilde{\mathcal{D}} | M, \mathcal{D}) = \{\mathbf{y}_{1:N}^s \sim p(\cdot | M, \boldsymbol{\theta}^s), \boldsymbol{\theta}^s \sim p(\boldsymbol{\theta} | \mathcal{D}, M)\} \quad (3.317)$$

These represent “plausible hallucinations” of the model. To assess the quality of our model, we can compute how “typical” our observed data \mathcal{D} is compared to the model’s hallucinations. To perform this comparison, we create one or more scalar **test statistics**, $\text{test}(\tilde{\mathcal{D}}^s)$, and compare them to the test statistics on the actual data, $\text{test}(\mathcal{D})$. These statistics should measure features of interest (since it will not, in general, be possible to capture every aspect of the data with a given model). If there is a large difference between the distribution of $\text{test}(\tilde{\mathcal{D}}^s)$ across different s and the value of $\text{test}(\mathcal{D})$, it suggests the model is not a good one. This approach called a **posterior predictive check** [Rub84].

3.9.1.1 Example: 1d Gaussian

To make things clearer, let us consider an example from [Gel+04]. In 1882, Newcomb measured the speed of light using a certain method and obtained $N = 66$ measurements, shown in Figure 3.25(a). There are clearly two outliers in the left tails, suggesting that the distribution is not Gaussian. Let us nonetheless fit a Gaussian to it. For simplicity, we will just compute the MLE, and use a plug-in approximation to the posterior predictive density:

$$p(\tilde{y} | \mathcal{D}) \approx \mathcal{N}(\tilde{y} | \hat{\mu}, \hat{\sigma}^2), \quad \hat{\mu} = \frac{1}{N} \sum_{n=1}^N y_n, \quad \hat{\sigma}^2 = \frac{1}{N} \sum_{n=1}^N (y_n - \hat{\mu})^2 \quad (3.318)$$

Let $\tilde{\mathcal{D}}^s$ be the s ’th dataset of size $N = 66$ sampled from this distribution, for $s = 1 : 1000$. The histogram of $\tilde{\mathcal{D}}^s$ for some of these samples is shown in Figure 3.25(b). It is clear that none of the samples contain the large negative examples that were seen in the real data. This suggests the model cannot capture the long tails present in the data. (We are assuming that these extreme values are scientifically interesting, and something we want the model to capture.)

A more formal way to test fit is to define a test statistic. Since we are interested in small values, let us use

$$\text{test}(\mathcal{D}) = \min\{y : y \in \mathcal{D}\} \quad (3.319)$$

The empirical distribution of $\text{test}(\tilde{\mathcal{D}}^s)$ for $s = 1 : 1000$ is shown in Figure 3.25(c). For the real data, $\text{test}(\mathcal{D}) = -44$, but the test statistics of the generated data, $\text{test}(\tilde{\mathcal{D}})$, are much larger. Indeed, we see that -44 is in the left tail of the predictive distribution, $p(\text{test}(\tilde{\mathcal{D}}) | \mathcal{D})$.

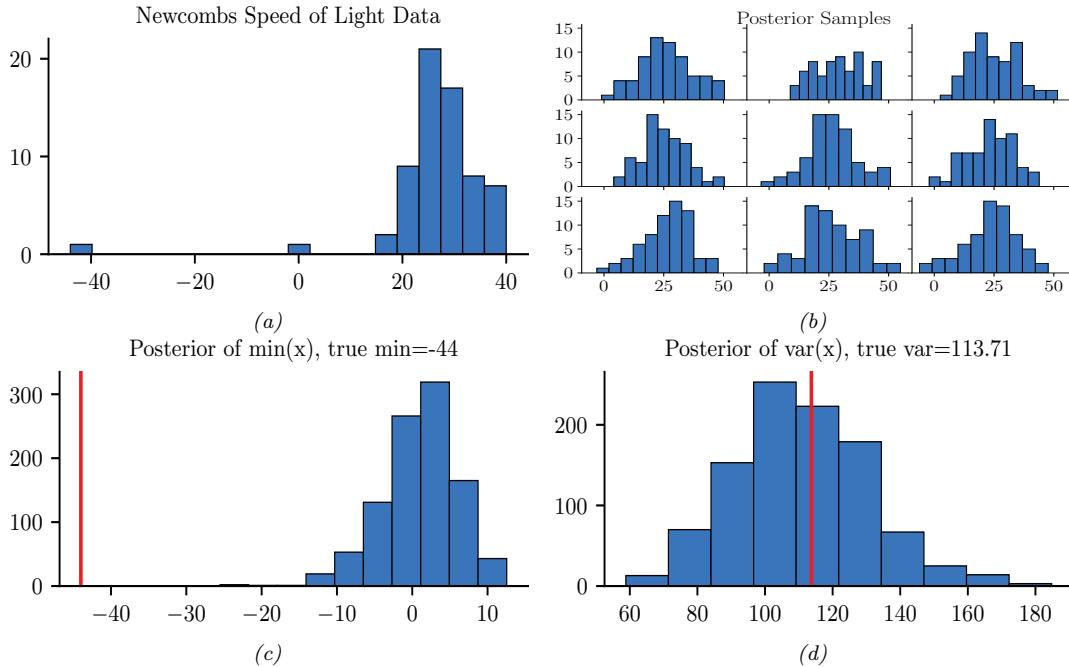


Figure 3.25: (a) Histogram of Newcomb’s data. (b) Histograms of data sampled from Gaussian model. (c) Histogram of test statistic on data sampled from the model, which represents $p(\text{test}(\tilde{\mathcal{D}}^s) | \mathcal{D})$, where $\text{test}(\mathcal{D}) = \min\{y \in \mathcal{D}\}$. The vertical line is the test statistic on the true data, $\text{test}(\mathcal{D})$. (d) Same as (c) except $\text{test}(\mathcal{D}) = \mathbb{V}\{y \in \mathcal{D}\}$. Generated by [newcomb_plugin_demo.ipynb](#).

3.9.1.2 Example: linear regression

When fitting conditional models, $p(\mathbf{y}|\mathbf{x})$, we will have a different prediction for each input \mathbf{x} . We can compare the predictive distribution $p(\mathbf{y}|\mathbf{x}_n)$ to the observed \mathbf{y}_n to detect places where the model does poorly.

As an example of this, we consider the “waffle divorce” dataset from [McE20, Sec 5.1]. This contains the divorce rate D_n , marriage rate M_n , and age A_n at first marriage for 50 different US states. We use a linear regression model to predict the divorce rate, $p(y = d | \mathbf{x} = (a, m)) = \mathcal{N}(d | \alpha + \beta_a a + \beta_m m, \sigma^2)$, using vague priors for the parameters. (In this example, we use a Laplace approximation to the posterior, discussed in Section 7.4.3.) We then compute the posterior predictive distribution $p(y|\mathbf{x}_n, \mathcal{D})$, which is a 1d Gaussian, and plot this vs each observed outcome y_n .

The result is shown in Figure 3.26. We see several outliers, some of which have been annotated. In particular, we see that both Idaho (ID) and Utah (UT) have a much lower divorce rate than predicted. This is because both of these states have an unusually large proportion of Mormons.

Of course, we expect errors in our predictive models. However, ideally the predictive error bars for the inputs where the model is wrong would be larger, rather than the model confidently making errors. In this case, the overconfidence arises from our incorrect use of a linear model.

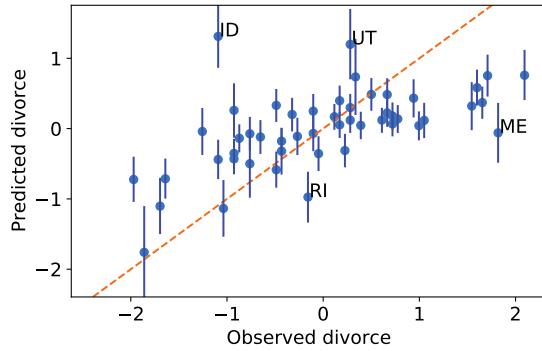


Figure 3.26: Posterior predictive distribution for divorce rate vs actual divorce rate for 50 US states. Both axes are standardized (i.e., z-scores). A few outliers are annotated. Adapted from Figure 5.5 of [McE20]. Generated by [linreg_divorce_ppc.ipynb](#).

3.9.2 Bayesian p-values

If some test statistic of the observed data, $\text{test}(\mathcal{D})$, occurs in the left or right tail of the predictive distribution, then it is very unlikely under the model. We can quantify this using a **Bayesian p-value**, also called a **posterior predictive p-value**:

$$p_B = \Pr(\text{test}(\tilde{\mathcal{D}}) \geq \text{test}(\mathcal{D}) | M, \mathcal{D}) \quad (3.320)$$

where M represents the model we are using, and $\tilde{\mathcal{D}}$ is a hypothetical future dataset. In contrast, a classical or frequentist **p-value** is defined as

$$p_C = \Pr(\text{test}(\tilde{\mathcal{D}}) \geq \text{test}(\mathcal{D}) | M) \quad (3.321)$$

where M represents the null hypothesis. The key difference is that the Bayesian compares what was observed to what one would expect after conditioning the model on the data, whereas the frequentist compares what was observed to the sampling distribution of the null hypothesis, which is independent of the data.

We can approximate the Bayesian p-value using Monte Carlo integration, as follows:

$$p_B = \int \mathbb{I}(\text{test}(\tilde{\mathcal{D}}) > \text{test}(\mathcal{D})) p(\tilde{\mathcal{D}}|\theta) p(\theta|\mathcal{D}) d\theta \approx \frac{1}{S} \sum_{s=1}^S \mathbb{I}(\text{test}(\tilde{\mathcal{D}}^s) > \text{test}(\mathcal{D})) \quad (3.322)$$

Any extreme value for p_B (i.e., a value near 0 or 1) means that the observed data is unlikely under the model, as assessed via test statistic test. However, if $\text{test}(\mathcal{D})$ is a sufficient statistic of the model, it is likely to be well estimated, and the p-value will be near 0.5. For example, in the speed of light example, if we define our test statistic to be the variance of the data, $\text{test}(\mathcal{D}) = \mathbb{V}\{y : y \in \mathcal{D}\}$, we get a p-value of 0.48. (See Figure 3.25(d).) This shows that the Gaussian model is capable of representing the variance in the data, even though it is not capable of representing the support (range) of the data.

The above example illustrates the very important point that we should not try to assess whether the data comes from a given model (for which the answer is nearly always that it does not), but rather, we should just try to assess whether the model captures the features we care about. See [Gel+04, ch.6] for a more extensive discussion of this topic.

3.10 Hypothesis testing

Suppose we have collected some coin tossing data, and we want to know if there if the coin is fair or not. Or, more interestingly, we have collected some clinical trial data, and want to know if there is a non-zero effect of the treatment on the outcome (e.g., different survival rates for the treatment and control groups). These kinds of problems can be solved using **hypothesis testing**. In the sections below, we summarize several common approaches to hypothesis testing.

3.10.1 Frequentist approach

In this section, we summarize the approach to hypothesis testing that is used in classical or frequentist statistics, which is known as **null hypothesis significance testing** or **NHST**. The basic idea is to define a binary decision rule of the form $\delta(\mathcal{D}) = \mathbb{I}(t(\mathcal{D}) \geq t^*)$, where $t(\mathcal{D})$ is some scalar **test statistic** derived from the data, and t^* is some **critical value**. If the test statistic exceeds the critical value, we **reject the null hypothesis**.

There is a large “zoo” of possible test statistics one can use (e.g., [Ken93] lists over 100 different tests), but a simple example is a **t-statistic**, defined as

$$t(\mathcal{D}) = \frac{\bar{x} - \mu}{\hat{\sigma}/\sqrt{N}} \tag{3.323}$$

where where \bar{x} is the empirical mean of \mathcal{D} , $\hat{\sigma}$ is the empirical standard deviation, N is the sample size, and μ is the **population mean**, corresponding to the mean value of the null hypothesis (often 0).

To compute the critical value t^* , we pick a **significance level** α , often 0.05, which controls the **type I error rate** of the decision procedure (i.e., the probability of accidentally rejecting the null hypothesis when it is true). We then find the value t^* whose tail probability, under the sampling distribution of the test statistic given the null hypothesis, matches the significance level:

$$p(t(\tilde{\mathcal{D}}) \geq t^* | H_0) = \alpha \tag{3.324}$$

This construction guarantees that $p(\delta(\tilde{\mathcal{D}}) = 1 | H_0) = \alpha$.

Rather than comparing $t(\mathcal{D})$ to t^* , a more common (but equivalent) approach is to compute the **p-value** of $t(\mathcal{D})$, which is defined in Equation (3.86). We can then reject the null hypothesis if $p < \alpha$.

Unfortunately, despite its widespread use, p-values and NHST have many problems, some of which are discussed in Section 3.3.5.2. We shall therefore avoid using this approach in this book.

3.10.2 Bayesian approach

In this section, we discuss the Bayesian approach to hypothesis testing. There are in fact two approaches, one based on model comparison using Bayes factors (Section 3.10.2.1), and one based on parameter estimation (Section 3.10.2.3).

Bayes factor $BF(1, 0)$	Interpretation
$BF < \frac{1}{100}$	Decisive evidence for M_0
$BF < \frac{1}{10}$	Strong evidence for M_0
$\frac{1}{10} < BF < \frac{1}{3}$	Moderate evidence for M_0
$\frac{1}{3} < BF < 1$	Weak evidence for M_0
$1 < BF < 3$	Weak evidence for M_1
$3 < BF < 10$	Moderate evidence for M_1
$BF > 10$	Strong evidence for M_1
$BF > 100$	Decisive evidence for M_1

Table 3.1: Jeffreys scale of evidence for interpreting Bayes factors.

3.10.2.1 Model comparison approach

Bayesian hypothesis testing is a special case of Bayesian model selection (discussed in Section 3.8.1) when we just have two models, commonly called the **null hypothesis**, M_0 , and the **alternative hypothesis**, M_1 . Let us define the **Bayes factor** as the ratio of marginal likelihoods:

$$B_{1,0} \triangleq \frac{p(\mathcal{D}|M_1)}{p(\mathcal{D}|M_0)} = \frac{p(M_1|\mathcal{D})}{p(M_0|\mathcal{D})} / \frac{p(M_1)}{p(M_0)} \quad (3.325)$$

(This is like a **likelihood ratio**, except we integrate out the parameters, which allows us to compare models of different complexity.) If $B_{1,0} > 1$ then we prefer model 1, otherwise we prefer model 0 (see Table 3.1).

We give a worked example of how to compute Bayes factors for a binomial test in Section 3.8.1.1. For examples of computing Bayes factors for more complex tests, see e.g. [Etz+18; Ly+20].

3.10.2.2 Improper priors cause problems for Bayes factors

Problems can arise when we use improper priors (i.e., priors that do not integrate to 1) for Bayesian model selection, even though such priors may be acceptable for other purposes, such as parameter inference. For example, consider testing the hypotheses $M_0 : \theta \in \Theta_0$ vs $M_1 : \theta \in \Theta_1$. The posterior probability of M_0 is given by

$$p(M_0|\mathcal{D}) = \frac{p(M_0)L_0}{p(M_0)L_0 + p(M_1)L_1} \quad (3.326)$$

where $L_i = p(\mathcal{D}|M_i) = \int_{\Theta_i} p(\mathcal{D}|\theta)p(\theta|M_i)d\theta$ is the marginal likelihood for model i .

Suppose (for simplicity) that $p(M_0) = p(M_1) = 0.5$, and we use a uniform but improper prior over the model parameters, $p(\theta|M_0) \propto c_0$ and $p(\theta|M_1) \propto c_1$. Define $\ell_i = \int_{\Theta_i} p(\mathcal{D}|\theta)d\theta$, so $L_i = c_i\ell_i$. Then

$$p(M_0|\mathcal{D}) = \frac{c_0\ell_0}{c_0\ell_0 + c_1\ell_1} = \frac{\ell_0}{\ell_0 + (c_1/c_0)\ell_1} \quad (3.327)$$

Thus the posterior (and hence Bayes factor) depends on the arbitrary constants c_0 and c_1 . This is known as the **marginalization paradox**. For this reason, we should avoid using improper priors

when performing Bayesian model selection. (However, if the same improper prior is used for common parameters that are shared between the two hypotheses, then the paradox does not arise.)

More generally, since the marginal likelihood is the likelihood averaged wrt the prior, results can be quite sensitive to the form of prior that is used. (See also Section 3.8.5, where we discuss conditional marginal likelihood.)

3.10.2.3 Parameter estimation approach

There are several drawbacks of the Bayesian hypothesis testing approach in Section 3.10.2.1, such as computational difficulty of computing the marginal likelihood (see Section 3.8.3), and the sensitivity to the prior (see Section 3.10.2.2). An alternative approach is to estimate the parameters of the model in the usual way, and then to see how much posterior probability is assigned to the parameter value corresponding to the null hypothesis. For example, to “test” if a coin is fair, we can first compute the posterior $p(\theta|\mathcal{D})$, and then we can evaluate the plausibility of the null hypothesis by computing $p(0.5 - \epsilon < \theta < 0.5 + \epsilon|\mathcal{D})$, where $(0.5 - \epsilon, 0.5 + \epsilon)$ is called the **region of practical equivalence** or **ROPE** [Kru15; KL17c]. This is not only computationally simpler, but is also allows us to quantify the effect size (i.e., the expected deviation of θ from the null value of 0.5), rather than merely accepting or rejecting a hypothesis. This approach is therefore called **Bayesian estimation**. We give some examples below, following <https://www.sumsar.net/blog/2014/01/bayesian-first-aid/>. (See also Section 3.10.3 for ways to perform more general tests usings GLMs.)

3.10.2.4 One sample test of a proportion (Binomial test)

Suppose we perform N coin tosses and observe y heads, where the frequency of heads is θ . We want to test the null hypothesis that $\theta = 0.5$. In frequentist statistics, we can use a **binomial test**. We now present a Bayesian alternative.

First we compute the posterior, $p(\theta|\mathcal{D}) \propto p(\theta)\text{Bin}(x|\theta, N)$. To do this, we need to specify a prior. We will use a noninformative prior. Following Section 3.5.2.1, the Jeffreys prior is $p(\theta) \propto \text{Beta}(\theta|\frac{1}{2}, \frac{1}{2})$, but [Lee04] argues that the uniform or flat prior, $p(\theta) \propto \text{Beta}(\theta|1, 1)$, is the least informative when we know that both heads and tails are possible. The posterior then becomes $p(\theta|\mathcal{D}) = \text{Beta}(\theta|y+1, N-y+1)$. From this, we can compute the credible interval $I = (\ell, u)$ using $\ell = P^{-1}(\alpha/2)$ and $u = P^{-1}(1-\alpha/2)$, where P is the cdf of the posterior. We can also easily compute the probability that the frequency exceeds the null value using

$$p(\theta > 0.5|\mathcal{D}) = \int_{0.5}^1 p(\theta|\mathcal{D})d\theta \quad (3.328)$$

We can compute this quantity using numerical integration or analytically [Coo05].

3.10.2.5 Two sample test of relative proportions (χ^2 test)

Now consider the setting where we have J groups, and in each group j we observe y_j successes in N_j trials. We denote the success rate by θ_j , and we are interested in testing the hypothesis that θ_j is the same for all the groups. In frequentist statistics, we can use a χ^2 test. Here we present a Bayesian alternative.

We will use an extension of Section 3.10.2.4, namely $y_j \sim \text{Bin}(\theta_j, N_j)$, where $\theta_j \sim \text{Beta}(1, 1)$, for $j = 1 : J$. To simplify notation, assume we $J = 2$ groups. The posterior is given by

$$p(\theta_1, \theta_2 | \mathcal{D}) = \text{Beta}(\theta_1 | y_1 + 1, N_1 - y_1 + 1) \text{Beta}(\theta_2 | y_2 + 1, N_2 - y_2 + 1) \quad (3.329)$$

We can then compute the posterior of the group difference, $\delta = \theta_1 - \theta_2$, using

$$p(\delta | \mathcal{D}) = \int_0^1 \int_0^1 \mathbb{I}(\delta = \theta_1 - \theta_2) p(\theta_1 | \mathcal{D}_1) p(\theta_2 | \mathcal{D}_2) \quad (3.330)$$

$$= \int_0^1 \text{Beta}(\theta_1 | y_1 + 1, N_1 - y_1 + 1) \text{Beta}(\theta_1 - \delta | y_2 + 1, N_2 - y_2 + 1) d\theta_1 \quad (3.331)$$

We can then use $p(\delta > 0 | \mathcal{D})$ to decide if the relative proportions between the two groups are significantly different or not.

3.10.2.6 One sample test of a mean (*t*-test)

Consider a dataset where we have N real-valued observations y_n which we assume come from a Gaussian, $y_n \sim \mathcal{N}(\mu, \sigma^2)$. We would like to test the hypothesis that $\mu = 0$. In frequentist statistics, the standard approach to this is to use a **t-test**, which is based on the sampling distribution of the standardized estimated mean. Here we develop a Bayesian alternative.

If we use a noninformative prior (which is a limiting case of the conjugate Gaussian-gamma prior), then the posterior for $p(\mu | \mathcal{D})$, after marginalizing out σ^2 , is the same as the sampling distribution of the MLE, $\hat{\mu}$, as we show in Section 15.2.3.2. In particular, both have a Student *t* distribution. Consequently, the Bayesian credible interval will be the same as the frequentist confidence interval in this simple setting.

However, a flat or noninformative prior for $p(\mu) \propto 1$ and $p(\sigma) \propto 1$ can give poor results, since we usually do not expect arbitrarily large values. According to [GHV20a], it is generally better to use **weakly informative priors**, whose hyperparameters can be derived from statistics of the data. For example, for the mean, we can use $p(\mu) = \mathcal{N}(\mu = 0, \sigma = 2.5\text{sd}(Y))$ (assuming the data is centered), and for the standard deviation, we can use $p(\sigma) = \text{Half-Student-t}(\mu = 0, \sigma = \text{sd}(y), \nu = 4)$.¹¹ These priors are no longer conjugate, but we can easily perform approximate posterior inference using MCMC or other algorithms discussed in Part II. We call this approach **BTT**, for “Bayesian *t*-test”.

[Kru13] proposes to use a Student likelihood $y_n \sim \mathcal{T}(\mu, \sigma, \nu)$ instead of a Gaussian likelihood, since it is more robust to outliers. He calls the method **BEST** method (“Bayesian Estimation Supersedes the *t*-test”), but we call it **robust BTT**. In addition to a different likelihood, robust BTT uses a different weakly informative prior, namely $\mu \sim \mathcal{N}(\mu = M_\mu, \sigma = S_\mu)$, $\sigma \in \text{Unif}(\sigma_{\text{low}}, \sigma_{\text{high}})$, and $\nu - 1 \sim \text{Expon}(1/29)$.¹²

11. This default prior is used by the Python **bambi** library [Cap+22], as well as the R **rstanarm** library (see <https://mc-stan.org/rstanarm/articles/priors.html>).

12. The prior for ν is an exponential distribution with mean 29 shifted 1 to the right, which keeps ν away from zero. According to [Kru13], “This prior was selected because it balances nearly normal distributions ($\nu > 30$) with heavy tailed distributions ($\nu < 30$)”. To avoid contamination from outliers, the prior for μ uses $M_\mu = M$, where M is the trimmed mean, and $S_\mu = 10^3 D$, where D is the mean absolute deviation. The prior for σ uses $\sigma_{\text{low}} = D/1000$ and $\sigma_{\text{high}} = D \times 1000$.

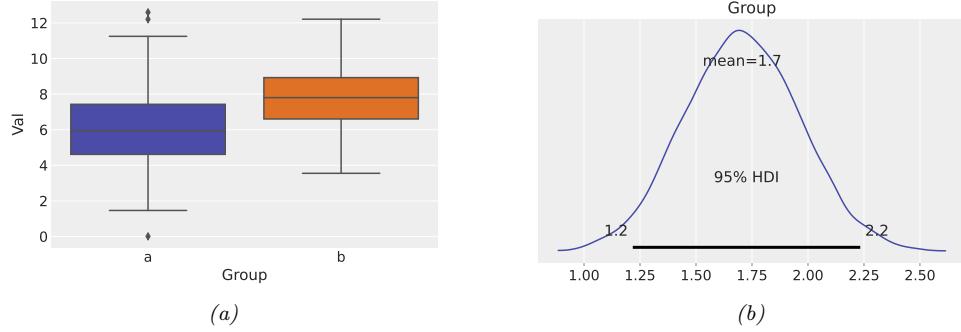


Figure 3.27: Illustration of Bayesian two-sample t-test. (a) Some synthetic data from two groups. (b) Posterior distribution of the difference, $p(\mu_2 - \mu_1 | \mathcal{D})$. Generated by `ttest_bambi.ipynb`.

3.10.2.7 Paired sample test of relative means (paired t-test)

Now suppose we have paired data from two groups, $\mathcal{D} = \{(y_{1n}, y_{2n}) : n = 1 : N\}$, where we assume $y_{jn} \sim \mathcal{N}(\mu_j, \sigma^2)$. We are interested in testing whether $\mu_1 = \mu_2$. A simpler alternative is to define $y_n = y_{2n} - y_{1n}$, which we model using $y_n \sim \mathcal{N}(\mu, \sigma^2)$. We can then test whether $\mu = 0$ using the t-test; this is called a **paired sample t-test**. In the Bayesian setting, we can just pass $\{y_n = y_{2n} - y_{1n}\}$ to the BTT procedure of Section 3.10.2.6.

3.10.2.8 Two sample test of relative means (two sample t-test)

In this section, we consider the setting in which we have two datasets, $\mathcal{D}_1 = \{y_{1n} \sim \mathcal{N}(\mu_1, \sigma_1^2) : n = 1 : N_1\}$ and $\mathcal{D}_2 = \{y_{2n} \sim \mathcal{N}(\mu_2, \sigma_2^2) : n = 1 : N_2\}$, and we want to test the null hypothesis that $\mu_1 = \mu_2$. If we assume $\sigma_1^2 = \sigma_2^2$, we can use a **two-sample t-test**, also called an **independent t-test** or **unpaired t-test**. If we allow the variance of the observations to vary by group, then we can use **Welch's t-test**.

In the Bayesian setting, we can tackle this by generalizing the BTT model of Section 3.10.2.6 to two groups by defining $y_{jn} \sim \mathcal{N}(\mu_j, \sigma_j^2)$, for $j = 1, 2$. (We can also use a robust likelihood.) Once we have specified the model, we can perform posterior inference in the usual way, and compute quantities such as $p(\mu_1 - \mu_2 > 0 | \mathcal{D})$. See Figure 3.27 for an example.

3.10.2.9 Testing a correlation coefficient

In this section, we consider the setting in which we have some data $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$, where (x, y) may be correlated with a **Pearson correlation coefficient** of ρ . We are interested in testing the null hypothesis that $\rho = 0$.

In the Bayesian setting, we can do this by generalizing the two-sample BTT approach of Section 3.10.2.8. Specifically, we assume

$$(x_n, y_n) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (3.332)$$

Y	X	P/N	Name	Model	Exact
\mathbb{R}	-	P	One-sample t-test	$y \sim \mathcal{N}(\mu, \sigma^2)$	✓
\mathbb{R}	-	N	Wilcoxon signed-ranked	$SR(y) \sim \mathcal{N}(\mu, \sigma^2)$	$N > 14$
(\mathbb{R}, \mathbb{R})	-	P	Paired-sample t-test	$y_2 - y_1 \sim \mathcal{N}(\mu, \sigma^2)$	✓
(\mathbb{R}, \mathbb{R})	-	N	Wilcoxon matched pairs	$SR(y_2 - y_1) \sim \mathcal{N}(\mu, \sigma^2)$	✓
\mathbb{R}	\mathbb{R}	P	Pearson correlation	$y \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma^2)$	✓
\mathbb{R}	\mathbb{R}	N	Spearman correlation	$R(y) \sim \mathcal{N}(\beta_0 + \beta_1 R(x), \sigma^2)$	$N > 10$
\mathbb{R}	$\{0, 1\}$	P	Two-sample t-test	$y \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma^2)$	✓
\mathbb{R}	$\{0, 1\}$	P	Welch's t-test	$y \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma_x^2)$	✓
\mathbb{R}	$\{0, 1\}$	N	Mann-Whitney U	$SR(y) \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma_x^2)$	$N > 11$
\mathbb{R}	$[J]$	P	One-way ANOVA	$y \sim \mathcal{N}(f_{aov}(x; \boldsymbol{\beta}), \sigma^2)$	✓
\mathbb{R}	$[J]$	N	Kruskal-Wallis	$R(y) \sim \mathcal{N}(f_{aov}(x; \boldsymbol{\beta}), \sigma^2)$	$N > 11$
\mathbb{R}	$[J] \times [K]$	N	Two-way ANOVA	$y \sim \mathcal{N}(f_{aov2}(x_1, x_2; \boldsymbol{\beta}), \sigma^2)$	✓

Table 3.2: Many common statistical tests are equivalent to performing inference for the parameters of simple linear models. Here P/N represents parametric vs nonparametric test; we approximate the latter by using the rank function $R(y)$ or the signed rank function $SR(y)$. The last column, labeled “exact”, specifies the sample size for which this approximation becomes accurate enough to be indistinguishable from the exact result. When the input variable is categorical, $x_1 \in [J]$, where $[J] = \{1, \dots, J\}$, we define the mean of the output using the analysis of variance function $f_{aov}(x_1, \boldsymbol{\beta})$. When we have two categorical inputs, $x_1 \in [J]$ and $x_2 \in [K]$, we use $f_{aov2}(x_1, x_2; \boldsymbol{\beta})$. Adapted from the crib sheet at <https://lindeloev.github.io/tests-as-linear/>.

where $\boldsymbol{\mu} = [\mu_1, \mu_2]$, and

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_a\sigma_2 & \sigma_2^2 \end{pmatrix} \quad (3.333)$$

We use the same (data-driven) priors for μ_j and σ_j , and use a uniform prior for the correlation, $p(\rho) = \text{Unif}(-1, 1)$, following [BMM00]. Once we have specified the model, we can perform posterior inference in the usual way, and compute quantities such as $p(\rho > 0 | \mathcal{D})$.

3.10.3 Common statistical tests correspond to inference in linear models

We have now seen many different tests, and it may be unclear what test to use when. Fortunately, [Lin19] points out that many of the most common tests can be represented exactly (or approximately) in terms of inference (either Bayesian or frequentist) about the parameters of a generalized linear model or GLM (see Chapter 15 for details on GLMs). This approach is easier to understand and more flexible, as discussed at length in e.g., [Kru15; GHV20b]. We summarize some of these results in Table 3.2 and the discussion below.

3.10.3.1 Approximating nonparametric tests using the rank transform

It is common to use “**nonparametric tests**”, which generalize standard tests to settings where the data do not necessarily follow a Gaussian or Student distribution. A simple way to approximate such tests is to replace the original data with its **order statistics**, and then to apply a standard parametric

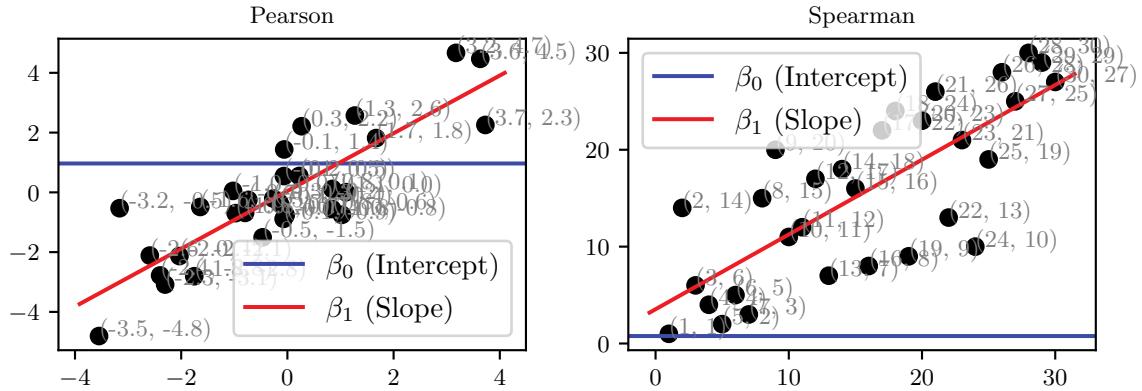


Figure 3.28: Illustration of 1d linear regression applied to some data (left) and its rank-transformed version (right). Generated by [linreg_rank_stats.ipynb](#).

test, as proposed in [CI81]. This gives a good approximation to the standard nonparametric tests for sample sizes of $N \geq 10$.

Concretely, we can compute a **rank transform**, in which the data points (assumed to be scalar) are sorted, and then replaced by their integer value in the ordering. For example, the rank transform of $\mathcal{D} = (3.6, 3.4, -5.0, 8.2)$ is $R(\mathcal{D}) = (3, 2, 1, 4)$. Alternatively we may use the **signed ranked**, which first sorts the values according to their absolute size, and then attaches the corresponding sign. For example, the signed rank transform of $\mathcal{D} = (3.6, 3.4, -5.0, 8.2)$ is $SR(\mathcal{D}) = (2, 1, -3, 4)$.

We can now easily fit a parametric model, such as a GLM, to the rank-transformed data, as illustrated in Figure 3.28. (In [Doo+17], they propose a Bayesian interpretation of this, where the order statistics are viewed as observations of an underlying latent continuous quantity, on which inference is performed.) We will use this trick in the sections below.

3.10.3.2 Metric-predicted variable on one or two groups (*t*-test)

Suppose we have some data $\mathcal{D} = \{y_n \sim \mathcal{N}(\mu, \sigma^2) : n = 1 : N\}$, and we are interested in testing the null hypothesis that $\mu = 0$. We can model this as a linear regression model with a constant input (bias term), and no covariates: $p(y_n | \boldsymbol{\theta}) = \mathcal{N}(y_n | \beta_0, \sigma^2)$, where $\beta_0 = \mu$. We can now perform inference on β_0 in the usual way for GLMs, and then perform hypothesis testing. This is equivalent to the **one sample t-test** discussed in Section 3.10.2.6. For a nonparametric version, we can transform the data using the signed rank transform, thus fitting $SR(y_n) \sim \mathcal{N}(\mu, \sigma^2)$. The results are very close to the **Wilcoxon signed-ranked test**.

Now suppose we have paired data from two groups, $\mathcal{D} = \{(y_{1n}, y_{2n}) : n = 1 : N\}$, where we assume $y_{jn} \sim \mathcal{N}(\mu_j, \sigma^2)$. We are interested in testing whether $\mu_1 = \mu_2$. A simpler alternative is to define $y_n = y_{2n} - y_{1n}$, which we model using $y_n \sim \mathcal{N}(\mu, \sigma^2)$. We can then test whether $\mu = 0$ using the **paired sample t-test**, discussed in Section 3.10.2.6. Alternatively we can do inference on $SR(y_n)$, to get the **Wilcoxon matched pairs** test.

To handle the setting in which we have unpaired data from two groups, we can represent the data as $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$, where $x_n \in \{0, 1\}$ represents whether the input belongs

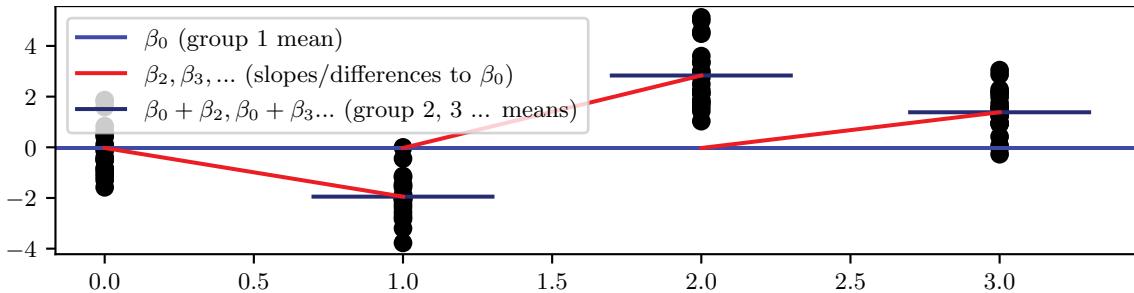


Figure 3.29: Illustration of one-way ANOVA with 4 groups. We are interested in testing whether the red lines have a slope of 0, meaning that all the groups have the same mean. Generated by [anova.ipynb](#).

to group 0 or group 1. We assume the data comes from the following linear regression model: $p(y_n|x_n) \sim \mathcal{N}(\beta_0 + \beta_1 x_n, \sigma^2)$. We can now perform inference on β in the usual way for GLMs, and then perform hypothesis testing. This is equivalent to the **two-sample t-test** discussed in Section 3.10.2.8. In the nonparametric setting, we can replace y with its signed ranked transform and use the model $SR(y) \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma^2)$. This is approximately the same as the **Mann-Whitney U test**.

3.10.3.3 Metric-predicted variable with metric predictors (correlation test)

In this section, we assume the data has the form $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$, where $x_n \in \mathbb{R}$ and $y_n \in \mathbb{R}$ are correlated with **Pearson correlation coefficient** of ρ . We are interested in testing the hypothesis that $\rho = 0$.

We can use a “bespoke” Bayesian approach as in Section 3.10.2.9. Alternatively, we can model this using simple linear regression, by writing $y_n \sim \mathcal{N}(\beta_0 + \beta_1 x, \sigma^2)$. If we scale the output Y so it has a standard deviation of 1, then we find that $\beta_1 = \rho$, as shown in [Mur22, Sec 11.2.3.3]. Thus we can use $p(\beta_1|\mathcal{D})$ to make inferences about ρ .

In the nonparametric setting, we compute the rank transform of x and y and then proceed as above. The **Spearman rank correlation coefficient** is the Pearson correlation coefficient on the rank-transformed data. While Pearson’s correlation is useful for assessing the strength of linear relationships, Spearman’s correlation can be used to assess general monotonic relationships, whether linear or not.

If we have multiple metric predictors (i.e., $\mathbf{x}_n \in \mathbb{R}^D$), we can use multiple linear regression instead of simple linear regression. We can then derive the posterior of the partial correlation coefficient from the posterior of the regression weights.

3.10.3.4 Metric-predicted variable with one nominal predictor (one-way ANOVA)

In this section, we consider the setting in which we have some data $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$, where $x_n \in \{1, \dots, J\}$ represents which group the input belongs. (Such a discrete categorical variable is often called a **factor**.) We assume the data comes from the following linear regression model: $p(y_n|x_n = j) \sim \mathcal{N}(\mu_j, \sigma^2)$. We are interested in testing the hypothesis that all the μ_j are the same. This is traditionally performed using a **one-way ANOVA test**, where ANOVA stands for “analysis

of variance". To derive a nonparametric test, we can first apply a rank transformation to y . This is similar to the **Kruskal-Wallis test**.

ANOVA assumes that the data are normally distributed, with a common (shared) variance, so that the sampling distribution of the **F -statistic** can be derived. We can write the corresponding model as a linear regression model, by using a dummy encoding of x_n , where $x_{n[j]} = \mathbb{I}(x_n = j)$. To avoid overparameterization (which can make the posterior unidentifiable), we drop the first level (this is known as **reduced rank encoding**). We can then write the model as

$$p(y_n|x_n; \boldsymbol{\theta}) \sim \mathcal{N}(f_{\text{aov}}(\mathbf{x}_n, \boldsymbol{\beta}), \sigma^2) \quad (3.334)$$

where we define the predicted mean using the ANOVA formula:

$$f_{\text{aov}}(\mathbf{x}, \boldsymbol{\beta}) = \beta_0 + \beta_2 x_{[2]} + \cdots + \beta_J x_{[J]} \quad (3.335)$$

We see that β_0 is the overall mean, and also corresponds to the value that will be used for level 1 of the factor (i.e., if $x_n = 1$). The other β_j terms represents deviations away from level 1. The null hypothesis corresponds to the assumption that $\beta_j = 0$ for all $j = 2 : J$.

A more symmetric formulation of the model is to write

$$f_{\text{aov}}(\mathbf{x}; \boldsymbol{\beta}) = \beta_0 + \beta_1 x_{[1]} + \beta_2 x_{[2]} + \cdots + \beta_J x_{[J]} \quad (3.336)$$

where β_0 is the **grand mean**, and where we impose the constraint that $\sum_{j=1}^J \beta_j = 0$. In this case we can interpret each β_j as the amount that group j deviates from the shared baseline β_0 . To satisfy this constraint, we can write the predicted mean as

$$f_{\text{aov}}(\mathbf{x}, \tilde{\boldsymbol{\beta}}) = \tilde{\beta}_0 + \sum_{j=1}^J \tilde{\beta}_j x_{[j]} = \underbrace{(\tilde{\beta}_0 + \bar{\beta})}_{\beta_0} + \sum_{j=1}^J \underbrace{(\tilde{\beta}_j - \bar{\beta})}_{\beta_j} x_{[j]} \quad (3.337)$$

where $\tilde{\beta}_j$ are the unconstrained parameters, and $\bar{\beta} = \frac{1}{J} \sum_{j=1}^J \tilde{\beta}_j$. This construction satisfies the constraint, since

$$\sum_{j=1}^J \beta_j = \sum_{j=1}^J \tilde{\beta}_j - \sum_{j=1}^J \bar{\beta} = J\bar{\beta} - J\bar{\beta} = 0 \quad (3.338)$$

In traditional ANOVA, we assume that the data are normally distributed, with a common (shared) variance. In a Bayesian setting, we are free to relax these assumptions. For example, we can use a different likelihood (e.g., Student) and we can allow each group to have its own variance, σ_j^2 , which can be reliably estimated using a hierarchical Bayesian model (see Section 3.6).

3.10.3.5 Metric-predicted variable with multiple nominal predictors (multi-way ANOVA)

In this section, we consider the setting in which we have G nominal predictors as input. To simplify notation, we assume we just have $G = 2$ groups. We assume the mean of y is given by

$$f_{\text{aov2}}(\mathbf{x}) = \mu + \sum_j \alpha_j x_{1,[j]} + \sum_k \beta_k x_{2,[k]} + \sum_{jk} \gamma_{jk} x_{1,[j]} x_{2,[k]} \quad (3.339)$$

	LH	RH	
Male	9	43	$N_1 = 52$
Female	4	44	$N_2 = 48$
Totals	13	87	100

Table 3.3: A 2×2 contingency table from http://en.wikipedia.org/wiki/Contingency_table.

where we impose the following sum-to-zero constraints

$$\sum_j \alpha_j = \sum_k \beta_k = \sum_j \gamma_{jk} = \sum_k \gamma_{jk} = 0 \quad (3.340)$$

We are interested in testing whether $\boldsymbol{\gamma} = \mathbf{0}$, meaning there is no interaction effect. This is traditionally done using a **two-way ANOVA test**. However, we can also use a Bayesian approach and just compute $p(\boldsymbol{\theta}|\mathcal{D})$.

3.10.3.6 Count predicted variable with nominal predictors (χ^2 test)

Consider a situation in which we observed two nominal values for each item measured. For example, the gender of a person (male or female) and whether they are left handed or right handed (LH or RH). If we count the number of outcomes of each type, we can represent the data as a $R \times C$ **contingency table**. See Table 3.3 for an example. We may be interested in testing the null hypothesis that there is no interaction effect between the two groups and the outcome (i.e., the two variables are independent). In frequentist statistics, this is often tackled using a **χ^2 -test**, which uses the sampling distribution of the χ^2 test statistic, defined as

$$\chi^2 = \sum_{r=1}^R \sum_{c=1}^C \frac{(O_{r,c} - E_{r,c})^2}{E_{r,c}} \quad (3.341)$$

where r indexes the rows, and c the columns, $O_{r,c}$ is the observed count in cell (r, c) , and $E_{rc} = N p_r p_c$ is the expected count, where $p_r = O_{r..}/N$ and $p_c = O_{..c}/N$ are the empirical marginal frequencies.

In the Bayesian approach, we can just modify the two-way ANOVA of Section 3.10.3.5, and replace the Gaussian distribution with a Poisson distribution. We also need to pass the predicted natural parameter through an exponential link, since a Poisson distribution requires that the rate parameter is non-negative. Thus the model becomes

$$p(y|\mathbf{x} = (r, c), \boldsymbol{\theta}) = \text{Poi}(y|\lambda_{r,c}) \quad (3.342)$$

$$\lambda_{rc} = \exp(\beta_0 + \beta_r + \beta_c + \beta_{r,c}) \quad (3.343)$$

We can now perform posterior inference in the usual way.

3.10.3.7 Non-metric predicted variables

If the output variable is categorical, $y_n \in \{1, \dots, C\}$, we can use logistic regression instead of linear regression (see e.g., Section 15.3.9). If the output is ordinal, we can use ordinal regression. If the output is a count variable, we can use Poisson regression. And so on. For more details on GLMs, see Chapter 15.

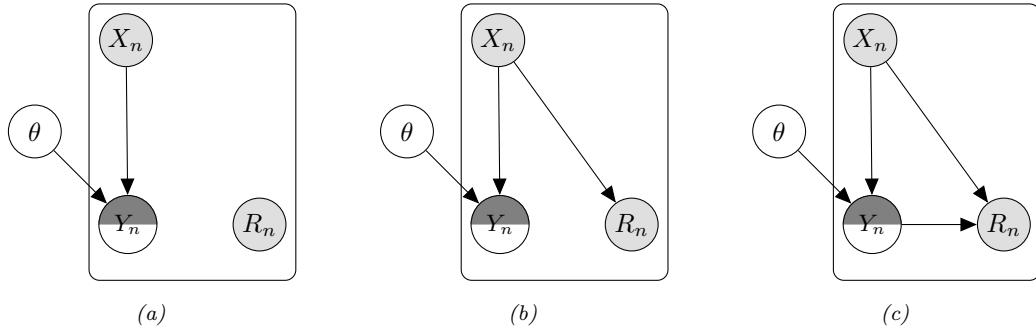


Figure 3.30: Graphical models to represent different patterns of missing data for conditional (discriminative) models. (a) Missing completely at random. (b) Missing at random. (c) Missing not at random. The semi-shaded y_n node is observed if $r_n = 1$ and is hidden otherwise. Adapted from Figure 2 of [SG02].

3.11 Missing data

Sometimes we may have **missing data**, in which parts of the data vector $\mathbf{X}_n \in \mathbb{R}^D$ may be unknown. (If we have a supervised problem, we append the labels to the feature vector.) We let $\mathbf{X}_{n,\text{mis}}$ represent the missing parts, and $\mathbf{X}_{n,\text{obs}}$ represent the observed parts. Since the reasons that data are missing may be informative (e.g., declining to answer a survey question such as “Do you have disease X?” may be an indication that the subject does in fact have it), we need to model the **missing data mechanism**. To do this, we introduce a random variable \mathbf{R}_n , to represent which parts of \mathbf{X}_n are “revealed” (observed) or not. Specifically, we set $\mathbf{R}_{n,\text{obs}} = 1$ for those indices (components) for which \mathbf{X}_n is observed, and set $\mathbf{R}_{n,\text{mis}} = 0$ for the other indices.

There are different kinds of assumptions we can make about the missing data mechanism, as discussed in [Rub76; LR87]. The strongest assumption is to assume the data is **missing completely at random** or **MCAR**. This means that $p(\mathbf{R}_n|\mathbf{X}_n) = p(\mathbf{R}_n)$, so the missingness does not depend on the hidden or observed features. A more realistic assumption is known as **missing at random** or **MAR**. This means that $p(\mathbf{R}_n|\mathbf{X}_n) = p(\mathbf{R}_n|\mathbf{X}_{n,\text{obs}})$, so the missingness does not depend on the hidden features, but may depend on the visible features. If neither of these assumptions hold, we say the data is **missing not at random** or **MNAR**.

Now consider the case of conditional, or discriminative models, in which we model the outcome \mathbf{y}_n given observed inputs \mathbf{x}_n using a model of the form $p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$. Since we are conditioning on \mathbf{x}_n , we assume it is always observed. However, the output labels may or may not be observed, depending on the value of r_n . For example, in **semi-supervised learning**, we have a combination of labeled data, $\mathcal{D}^L = \{(\mathbf{x}_n, \mathbf{y}_n)\}$, and unlabeled data, $\mathcal{D}^U = \{(\mathbf{x}_n)\}$ [CSZ06].

The 3 missing data scenarios for the discriminative setting are shown in Figure 3.30, using graphical model notation (see [MPT13] for details). In the MCAR and MAR cases, we see that we can just ignore the unlabeled data with missing outputs, since the unknown model parameters $\boldsymbol{\theta}$ are unaffected by \mathbf{y}_n if it is a hidden leaf node. However, in the MNAR case, we see that $\boldsymbol{\theta}$ depends on \mathbf{y}_n , even it is hidden, since the value of \mathbf{y}_n is assumed to affect the probability of r_n , which is always observed. In such cases, to fit the model, we need to **impute** the missing values, by using methods such as EM (see Section 6.5.3).

Now consider the case where we use a joint or generative model of the form $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y})p(\mathbf{x}|\mathbf{y})$, instead of a discriminative model of the form $p(\mathbf{y}|\mathbf{x})$.¹³ In this case, the unlabeled data can be useful for learning even in the MCAR and MAR scenarios, since $\boldsymbol{\theta}$ now depends on both \mathbf{x} and \mathbf{y} . In particular, information about $p(\mathbf{x})$ can be informative about $p(\mathbf{y}|\mathbf{x})$. See e.g., [CSZ06] for details.

¹³. In [Sch+12a], they call a model of the form $p(\mathbf{y}|\mathbf{x})$ a “**causal classifier**”, since the features cause the labels, and a model of the form $p(\mathbf{x}|\mathbf{y})$ an “**anti-causal classifier**”, since the features are caused by the labels.

4 Graphical models

4.1 Introduction

I basically know of two principles for treating complicated systems in simple ways: the first is the principle of modularity and the second is the principle of abstraction. I am an apologist for computational probability in machine learning because I believe that probability theory implements these two principles in deep and intriguing ways — namely through factorization and through averaging. Exploiting these two mechanisms as fully as possible seems to me to be the way forward in machine learning. — Michael Jordan, 1997 (quoted in [Fre98]).

Probabilistic graphical models (PGMs) provide a convenient formalism for defining joint distributions on sets of random variables. In such graphs, the nodes represent random variables, and the (lack of) edges represent **conditional independence (CI)** assumptions between these variables. A better name for these models would be “independence diagrams”, but the term “graphical models” is now entrenched.

There are several kinds of graphical model, depending on whether the graph is directed, undirected, or some combination of directed and undirected, as we discuss in the sections below. More details on graphical models can be found in e.g., [KF09a].

4.2 Directed graphical models (Bayes nets)

In this section, we discuss directed probabilistic graphical models, or **DPGM**, which are based on **directed acyclic graphs** or **DAGs** (graphs that do not have any directed cycles). PGMs based on a DAG are often called **Bayesian networks** or **Bayes nets** for short; however, there is nothing inherently “Bayesian” about Bayesian networks: they are just a way of defining probability distributions. They are also sometimes called **belief networks**. The term “belief” here refers to subjective probability. However, the probabilities used in these models are no more (and no less) subjective than in any other kind of probabilistic model.

4.2.1 Representing the joint distribution

The key property of a DAG is that the nodes can be ordered such that parents come before children. This is called a **topological ordering**. Given such an order, we define the **ordered Markov property** to be the assumption that a node is conditionally independent of all its predecessors in

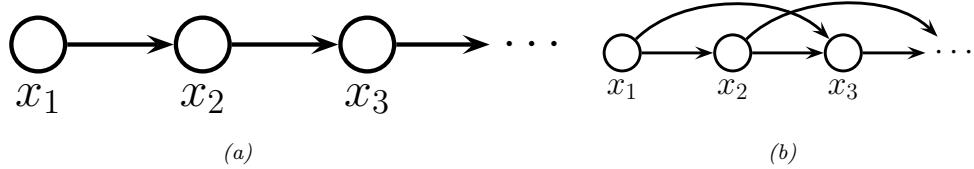


Figure 4.1: Illustration of first and second order Markov models.

the ordering given its parents, i.e.,

$$x_i \perp \mathbf{x}_{\text{pred}(i) \setminus \text{pa}(i)} | \mathbf{x}_{\text{pa}(i)} \quad (4.1)$$

where $\text{pa}(i)$ are the parents of node i , and $\text{pred}(i)$ are the predecessors of node i in the ordering. Consequently, we can represent the joint distribution as follows (assuming we use node ordering $1 : N_G$):

$$p(\mathbf{x}_{1:N_G}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)\dots p(x_{N_G}|x_1, \dots, x_{N_G-1}) = \prod_{i=1}^{N_G} p(x_i|\mathbf{x}_{\text{pa}(i)}) \quad (4.2)$$

where $p(x_i | \mathbf{x}_{\text{pa}(i)})$ is the **conditional probability distribution** or **CPD** for node i . (The parameters of this distribution are omitted from the notation for brevity.)

The key advantage of the representation used in Equation (4.2) is that the number of parameters used to specify the joint distribution is substantially less, by virtue of the conditional independence assumptions that we have encoded in the graph, than an unstructured joint distribution. To see this, suppose all the variables are discrete and have K states each. Then an unstructured joint distribution needs $O(K^{N_G})$ parameters to specify the probability of every configuration. By contrast, with a DAG in which each node has at most N_P parents, we only need $O(N_G K^{N_P+1})$ parameters, which can be exponentially fewer if the DAG is sparse.

We give some examples of DPGM's in Section 4.2.2, and in Section 4.2.4, we discuss how to read off other conditional independence properties from the graph.

4.2.2 Examples

In this section, we give several examples of models that can be usefully represented as DPGM's.

4.2.2.1 Markov chains

We can represent the conditional independence assumptions of a first-order Markov model using the chain-structured DPGM shown in Figure 4.1(a). Consider a variable at a single time step t , which we call the “present”. From the diagram, we see that information cannot flow from the past, $\mathbf{x}_{1:t-1}$, to the future, $\mathbf{x}_{t+1:T}$, except via the present, x_t . (We formalize this in Section 4.2.4.) This means that the x_t is a sufficient statistic for the past, so the model is first-order Markov. This implies that the corresponding joint distribution can be written as follows:

$$p(\mathbf{x}_{1:T}) = p(x_1)p(x_2|x_1)p(x_3|x_2)\cdots p(x_T|x_{T-1}) = p(x_1) \prod_{t=2}^T p(x_t|\mathbf{x}_{1:t-1}) \quad (4.3)$$

For discrete random variables, we can represent corresponding CPDs, $p(x_t = k|x_{t-1} = j)$, as a 2d table, known as a **conditional probability table** or **CPT**, $p(x_t = k|x_{t-1} = j) = \theta_{jk}$, where $0 \leq \theta_{jk} \leq 1$ and $\sum_{k=1}^K \theta_{jk} = 1$ (i.e., each row sums to 1).

The first-order Markov assumption is quite restrictive. If we want to allow for dependencies two steps into the past, we can create a Markov model of order 2. This is shown in Figure 4.1(b). The corresponding joint distribution has the form

$$p(\mathbf{x}_{1:T}) = p(x_1, x_2)p(x_3|x_1, x_2)p(x_4|x_2, x_3) \cdots p(x_T|x_{T-2}, x_{T-1}) = p(x_1, x_2) \prod_{t=3}^T p(x_t|\mathbf{x}_{t-2:t-1}) \quad (4.4)$$

As we increase the order of the Markov model, we need to add more edges. In the limit, the DAG becomes fully connected (subject to being acyclic), as shown in Figure 22.1. However, in this case, there are no useful conditional independencies, so the graphical model has no value.

4.2.2.2 The “student” network

Figure 4.2 shows a model for capturing the inter dependencies between 5 discrete random variables related to a hypothetical student taking a class: D = difficulty of class (easy, hard), I = intelligence (low, high), G = grade (A, B, C), S = SAT score (bad, good), L = letter of recommendation (bad, good). (This is a simplification of the “**student network**” from [KF09a, p.281].) The chain rule tells us that we can represent the joint as follows:

$$p(D, I, G, L, S) = p(L|S, G, D, I) \times p(S|G, D, I) \times p(G|D, I) \times p(D|I) \times p(I) \quad (4.5)$$

where we have ordered the nodes topologically as I, D, G, S, L. Note that L is conditionally independent of all the other nodes earlier in this ordering given its parent G, so we can replace $p(L|S, G, D, I)$ by $p(L|G)$. We can simplify the other terms in a similar way to get

$$p(D, I, G, L, S) = p(L|G) \times p(S|I) \times p(G|D, I) \times p(D|I) \times p(I) \quad (4.6)$$

The ability to simplify a joint distribution in a product of small local pieces is the key idea behind graphical models.

In addition to the graph structure, we need to specify the conditional probability distributions (CPDs) at each node. For discrete random variables, we can represent the CPD as a table, which means we have a separate row (i.e., a separate categorical distribution) for each **conditioning case**, i.e., for each combination of parent values. We can represent the i 'th CPT as follows:

$$\theta_{ijk} \triangleq p(x_i = k|\mathbf{x}_{\text{pa}(i)} = j) \quad (4.7)$$

The matrix $\boldsymbol{\theta}_{i,:,:}$ is a **row stochastic matrix**, that satisfies the properties $0 \leq \theta_{ijk} \leq 1$ and $\sum_{k=1}^{K_i} \theta_{ijk} = 1$ for each row j . Here i indexes nodes, $i \in [N_G]$; k indexes node states, $k \in [K_i]$, where K_i is the number of states for node i ; and j indexes joint parent states, $j \in [J_i]$, where $J_i = \prod_{p \in \text{pa}(i)} K_p$.

The CPTs for the student network are shown next to each node in Figure 4.2. For example, we see that if the class is hard ($D = 1$) and the student has low intelligence ($I = 0$), the distribution over grades A, B, and C we expect is $p(G|D = 1, I = 0) = [0.05, 0.25, 0.7]$; but if the student is intelligent, we get $p(G|D = 1, I = 1) = [0.5, 0.3, 0.2]$.

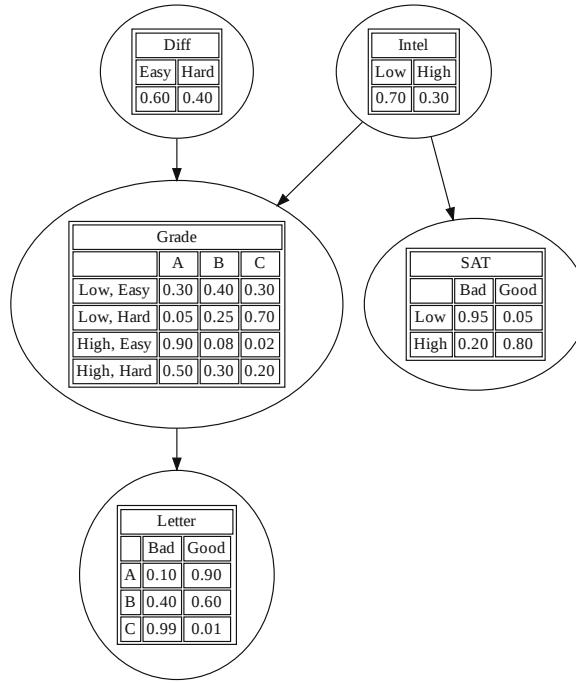


Figure 4.2: The (simplified) student network. “Diff” is the difficulty of the class. “Intel” is the intelligence of the student. “Grade” is the grade of the student in this class. “SAT” is the score of the student on the SAT exam. “Letter” is whether the teacher writes a good or bad letter of recommendation. The circles (nodes) represent random variables, the edges represent direct probabilistic dependencies. The tables inside each node represent the conditional probability distribution of the node given its parents. Generated by [student_pgm.ipynb](#).

The number of parameters in a CPT is $O(K^{p+1})$, where K is the number of states per node, and p is the number of parents. Later we will consider more parsimonious representations, with fewer learnable parameters. (We discuss parameter learning in Section 4.2.7.)

Once we have specified the model, we can use it to answer probabilistic queries, as we discuss in Section 4.2.6. As an example, suppose we observe that the student gets a grade of C. The posterior probability that the student is intelligent is just $p(I = \text{High}|G = C) = 0.08$, as shown in Figure 4.8. However, now suppose we also observe that the student gets a good SAT score. Now the posterior probability that the student is intelligent has jumped to $p(I = \text{High}|G = C, S = \text{Good}) = 0.58$, since we can explain the C grade by inferring it was a difficult class (indeed, we find $p(D = \text{Hard}|G = C, S = \text{Good}) = 0.76$). This negative mutual interaction between multiple causes of some observations is called the **explaining away** effect, also known as **Berkson’s paradox** (see Section 4.2.4.2 for details).

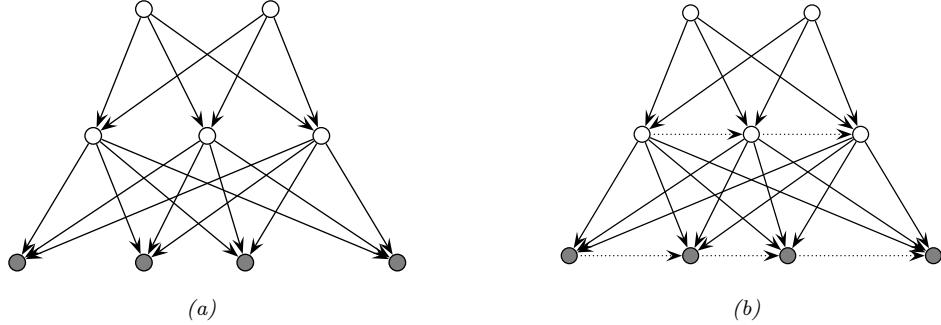


Figure 4.3: (a) Hierarchical latent variable model with 2 layers. (b) Same as (a) but with autoregressive connections within each layer. The observed \mathbf{x} variables are the shaded leaf nodes at the bottom. The unshaded nodes are the hidden \mathbf{z} variables.

4.2.2.3 Sigmoid belief nets

In this section, we consider a **deep generative model** of the form shown in Figure 4.3a. This corresponds to the following joint distribution:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z}_2)p(\mathbf{z}_1|\mathbf{z}_2)p(\mathbf{x}|\mathbf{z}_1) = \prod_{k=1}^{K_2} p(z_{2,k}) \prod_{k=1}^{K_1} p(z_{1,k}|\mathbf{z}_2) \prod_{d=1}^D p(x_d|\mathbf{z}_1) \quad (4.8)$$

where \mathbf{x} denotes the visible leaf nodes, and \mathbf{z}_ℓ denotes the hidden internal nodes. (We assume there are K_ℓ hidden nodes at level ℓ , and D visible leaf nodes.)

Now consider the special case where all the latent variables are binary, and all the latent CPDs are logistic regression models. That is,

$$p(\mathbf{z}_\ell|\mathbf{z}_{\ell+1}, \boldsymbol{\theta}) = \prod_{k=1}^{K_\ell} \text{Ber}(z_{\ell,k}|\sigma(\mathbf{w}_{\ell,k}^\top \mathbf{z}_{\ell+1})) \quad (4.9)$$

where $\sigma(u) = 1/(1 + e^{-u})$ is the sigmoid (logistic) function. The result is called a **sigmoid belief net** [Nea92].

At the bottom layer, $p(\mathbf{x}|\mathbf{z}_1, \boldsymbol{\theta})$, we use whatever observation model is appropriate for the type of data we are dealing with. For example, for real valued data, we might use

$$p(\mathbf{x}|\mathbf{z}_1, \boldsymbol{\theta}) = \prod_{d=1}^D \mathcal{N}(x_d | \mathbf{w}_{1,d,\mu}^\top \mathbf{z}_1, \exp(\mathbf{w}_{1,d,\sigma}^\top \mathbf{z}_1)) \quad (4.10)$$

where $\mathbf{w}_{1,d,\mu}$ are the weights that control the mean of the d 'th output, and $\mathbf{w}_{1,d,\sigma}$ are the weights that control the variance of the d 'th output.

We can also add directed connections between the hidden variables within a layer, as shown in Figure 4.3b. This is called a **deep autoregressive network** or **DARN** model [Gre+14], which combines ideas from latent variable modeling and autoregressive modeling.

We discuss other forms of hierarchical generative models in Chapter 21.

4.2.3 Gaussian Bayes nets

Consider a DPGM where all the variables are real-valued, and all the CPDs have the following form, known as a **linear Gaussian CPD**:

$$p(x_i | \mathbf{x}_{\text{pa}(i)}) = \mathcal{N}(x_i | \mu_i + \mathbf{w}_i^\top \mathbf{x}_{\text{pa}(i)}, \sigma_i^2) \quad (4.11)$$

As we show below, multiplying all these CPDs together results in a large joint Gaussian distribution of the form $p(\mathbf{x}) = \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\mathbf{x} \in \mathbb{R}^{N_G}$. This is called a **directed Gaussian graphical model** or a **Gaussian Bayes net**.

We now explain how to derive $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, following [SK89, App. B]. For convenience, we rewrite the CPDs in the following form:

$$x_i = \mu_i + \sum_{j \in \text{pa}(i)} w_{i,j}(x_j - \mu_j) + \sigma_i z_i \quad (4.12)$$

where $z_i \sim \mathcal{N}(0, 1)$, σ_i is the conditional standard deviation of x_i given its parents, $w_{i,j}$ is the strength of the $j \rightarrow i$ edge, and μ_i is the local mean.¹

It is easy to see that the global mean is just the concatenation of the local means, $\boldsymbol{\mu} = (\mu_1, \dots, \mu_{N_G})$. We now derive the global covariance, $\boldsymbol{\Sigma}$. Let $\mathbf{S} \triangleq \text{diag}(\boldsymbol{\sigma})$ be a diagonal matrix containing the standard deviations. We can rewrite Equation (4.12) in matrix-vector form as follows:

$$(\mathbf{x} - \boldsymbol{\mu}) = \mathbf{W}(\mathbf{x} - \boldsymbol{\mu}) + \mathbf{S}\mathbf{z} \quad (4.13)$$

where \mathbf{W} is the matrix of regression weights. Now let \mathbf{e} be a vector of noise terms: $\mathbf{e} \triangleq \mathbf{S}\mathbf{z}$. We can rearrange this to get $\mathbf{e} = (\mathbf{I} - \mathbf{W})(\mathbf{x} - \boldsymbol{\mu})$. Since \mathbf{W} is lower triangular (because $w_{j,i} = 0$ if $j < i$ in the topological ordering), we have that $\mathbf{I} - \mathbf{W}$ is lower triangular with 1s on the diagonal. Hence

$$\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_{N_G} \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ -w_{2,1} & 1 & & & \\ -w_{3,2} & -w_{3,1} & 1 & & \\ \vdots & & & \ddots & \\ -w_{N_G,1} & -w_{N_G,2} & \dots & -w_{N_G,N_G-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \\ \vdots \\ x_{N_G} - \mu_{N_G} \end{pmatrix} \quad (4.14)$$

Since $\mathbf{I} - \mathbf{W}$ is always invertible, we can write

$$\mathbf{x} - \boldsymbol{\mu} = (\mathbf{I} - \mathbf{W})^{-1} \mathbf{e} \triangleq \mathbf{U}\mathbf{e} = \mathbf{U}\mathbf{S}\mathbf{z} \quad (4.15)$$

where we defined $\mathbf{U} = (\mathbf{I} - \mathbf{W})^{-1}$. Hence the covariance is given by

$$\boldsymbol{\Sigma} = \text{Cov}[\mathbf{x}] = \text{Cov}[\mathbf{x} - \boldsymbol{\mu}] = \text{Cov}[\mathbf{U}\mathbf{S}\mathbf{z}] = \mathbf{U}\mathbf{S} \text{Cov}[\mathbf{z}] \mathbf{S}\mathbf{U}^\top = \mathbf{U}\mathbf{S}^2\mathbf{U}^\top \quad (4.16)$$

since $\text{Cov}[\mathbf{z}] = \mathbf{I}$.

1. If we do not subtract off the parent's mean (i.e., if we use $x_i = \mu_i + \sum_{j \in \text{pa}(i)} w_{i,j}x_j + \sigma_i z_i$), the derivation of $\boldsymbol{\Sigma}$ is much messier, as can be seen by looking at [Bis06, p370].

4.2.4 Conditional independence properties

We write $\mathbf{x}_A \perp_G \mathbf{x}_B | \mathbf{x}_C$ if A is conditionally independent of B given C in the graph G . (We discuss how to determine whether such a CI property is implied by a given graph in the sections below.) Let $I(G)$ be the set of all such CI statements encoded by the graph, and $I(p)$ be the set of all such CI statements that hold true in some distribution p . We say that G is an **I-map** (independence map) for p , or that p is **Markov** wrt G , iff $I(G) \subseteq I(p)$. In other words, the graph is an I-map if it does not make any assertions of CI that are not true of the distribution. This allows us to use the graph as a safe proxy for p when reasoning about p 's CI properties. This is helpful for designing algorithms that work for large classes of distributions, regardless of their specific numerical parameters. Note that the fully connected graph is an I-map of all distributions, since it makes no CI assertions at all, as we show below. We therefore say G is a **minimal I-map** of p if G is an I-map of p , and if there is no $G' \subseteq G$ which is an I-map of p .

We now turn to the question of how to derive $I(G)$, i.e., which CI properties are entailed by a DAG.

4.2.4.1 Global Markov properties (d-separation)

We say an *undirected path* P is **d-separated** by a set of nodes C (containing the evidence) iff at least one of the following conditions hold:

1. P contains a chain or **pipe**, $s \rightarrow m \rightarrow t$ or $s \leftarrow m \leftarrow t$, where $m \in C$
2. P contains a tent or **fork**, $s \swarrow^m \searrow t$, where $m \in C$
3. P contains a **collider** or **v-structure**, $s \searrow_m \swarrow t$, where m is not in C and neither is any descendant of m .

Next, we say that a *set of nodes* A is d-separated from a different set of nodes B given a third observed set C iff each undirected path from every node $a \in A$ to every node $b \in B$ is d-separated by C . Finally, we define the CI properties of a DAG as follows:

$$\mathbf{X}_A \perp_G \mathbf{X}_B | \mathbf{X}_C \iff A \text{ is d-separated from } B \text{ given } C \quad (4.17)$$

This is called the (directed) **global Markov property**.

The **Bayes ball algorithm** [Sha98] is a simple way to see if A is d-separated from B given C , based on the above definition. The idea is this. We “shade” all nodes in C , indicating that they are observed. We then place “balls” at each node in A , let them “bounce around” according to some rules, and then ask if any of the balls reach any of the nodes in B . The three main rules are shown in Figure 4.4. Notice that balls can travel opposite to edge directions. We see that a ball can pass through a chain, but not if it is shaded in the middle. Similarly, a ball can pass through a fork, but not if it is shaded in the middle. However, a ball cannot pass through a v-structure, unless it is shaded in the middle.

We can justify the 3 rules of Bayes ball as follows. First consider a chain structure $X \rightarrow Y \rightarrow Z$, which encodes

$$p(x, y, z) = p(x)p(y|x)p(z|y) \quad (4.18)$$

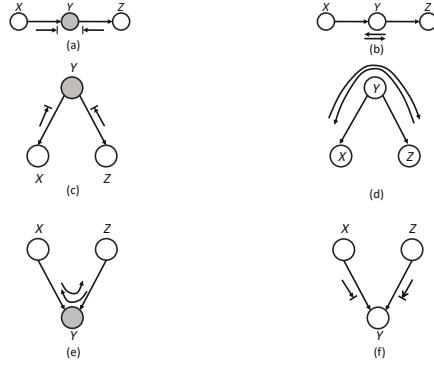


Figure 4.4: Bayes ball rules. A shaded node is one we condition on. If there is an arrow hitting a bar, it means the ball cannot pass through; otherwise the ball can pass through.

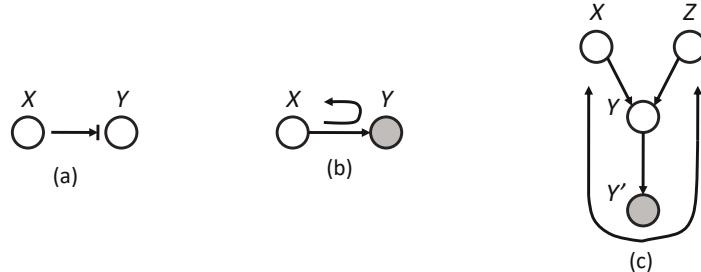


Figure 4.5: (a-b) Bayes ball boundary conditions. (c) Example of why we need boundary conditions. Y' is an observed child of Y , rendering Y “effectively observed”, so the ball bounces back up on its way from X to Z .

When we condition on y , are x and z independent? We have

$$p(x, z|y) = \frac{p(x, y, z)}{p(y)} = \frac{p(x)p(y|x)p(z|y)}{p(y)} = \frac{p(x, y)p(z|y)}{p(y)} = p(x|y)p(z|y) \quad (4.19)$$

and therefore $X \perp Z | Y$. So observing the middle node of chain breaks it in two (as in a Markov chain).

Now consider the tent structure $X \leftarrow Y \rightarrow Z$. The joint is

$$p(x, y, z) = p(y)p(x|y)p(z|y) \quad (4.20)$$

When we condition on y , are x and z independent? We have

$$p(x, z|y) = \frac{p(x, y, z)}{p(y)} = \frac{p(y)p(x|y)p(z|y)}{p(y)} = p(x|y)p(z|y) \quad (4.21)$$

and therefore $X \perp Z | Y$. So observing a root node separates its children (as in a naive Bayes classifier: see Section 4.2.8.2).

X	Y	Z
D	I	
D	I	S
D	S	
D	S	I
D	S	L, I
D	S	G, I
D	S	G, L, I
D	L	G
D	L	G, S
D	L	G, I
D	L	I, G, S

Table 4.1: Conditional independence relationships implied by the student DAG (Figure 4.2). Each line has the form $X \perp Y|Z$. Generated by [student_pgm.ipynb](#).

Finally consider a v-structure $X \rightarrow Y \leftarrow Z$. The joint is

$$p(x, y, z) = p(x)p(z)p(y|x, z) \quad (4.22)$$

When we condition on y , are x and z independent? We have

$$p(x, z|y) = \frac{p(x)p(z)p(y|x, z)}{p(y)} \quad (4.23)$$

so $X \not\perp Z|Y$. However, in the unconditional distribution, we have

$$p(x, z) = p(x)p(z) \quad (4.24)$$

so we see that X and Z are marginally independent. So we see that conditioning on a common child at the bottom of a v-structure makes its parents become dependent. This important effect is called **explaining away, inter-causal reasoning**, or **Berkson's paradox** (see Section 4.2.4.2 for a discussion).

Finally, Bayes ball also needs the “boundary conditions” shown in Figure 4.5(a-b). These rules say that a ball hitting a hidden leaf stops, but a ball hitting an observed leaf “bounces back”. To understand where this rule comes from, consider Figure 4.5(c). Suppose Y' is a (possibly noisy) copy of Y . If we observe Y' , we effectively observe Y as well, so the parents X and Z have to compete to explain this. So if we send a ball down $X \rightarrow Y \rightarrow Y'$, it should “bounce back” up along $Y' \rightarrow Y \rightarrow Z$, in order to pass information between the parents. However, if Y and *all* its children are hidden, the ball does not bounce back.

As an example of the CI statements encoded by a DAG, Table 4.1 shows some properties that follow from the student network in Figure 4.2.

4.2.4.2 Explaining away (Berkson's paradox)

In this section, we give some examples of the **explaining away** phenomenon, also called **Berkson's paradox**.

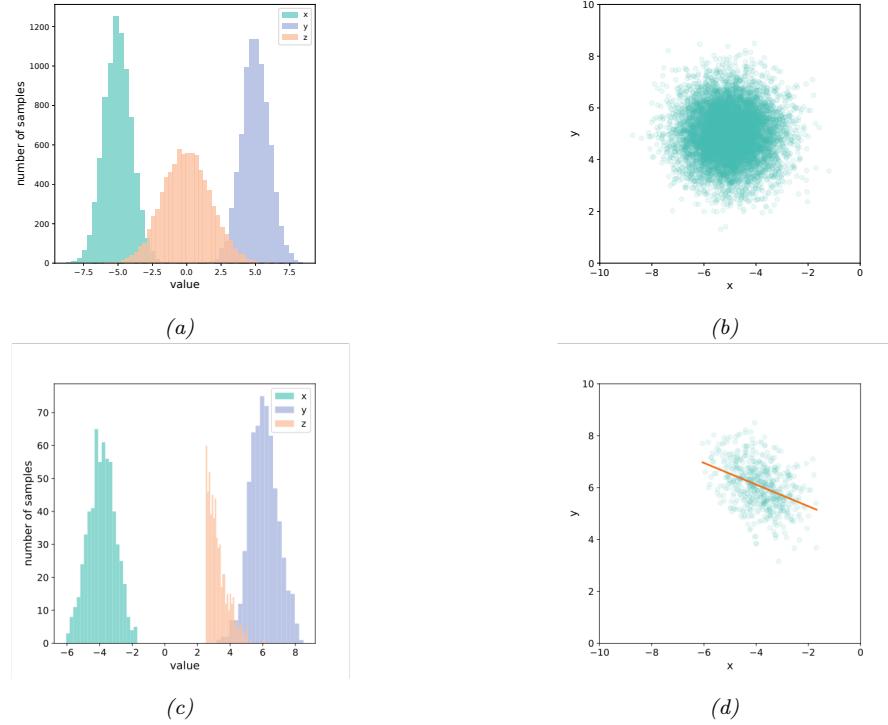


Figure 4.6: Samples from a jointly Gaussian DPGM, $p(x, y, z) = \mathcal{N}(x| -5, 1)\mathcal{N}(y|5, 1)\mathcal{N}(z|x + y, 1)$. (a) Unconditional marginal distributions, $p(x)$, $p(y)$, $p(z)$. (b) Unconditional joint distribution, $p(x, y)$. (c) Conditional marginal distribution, $p(x|z > 2.5)$, $p(y|z > 2.5)$, $p(z|z > 2.5)$. (d) Conditional joint distribution, $p(x, y|z > 2.5)$. Adapted from [Clo20]. Generated by [berksons_gaussian.ipynb](#).

As a simple example (from [PM18b, p198]), consider tossing two coins 100 times. Suppose you only record the outcome of the experiment if at least one coin shows up heads. You should expect to record about 75 entries. You will see that every time coin 1 is recorded as tails, coin 2 will be recorded as heads. If we ignore the way in which the data was collected, we might infer from the fact that coins 1 and 2 are correlated that there is a hidden common cause. However, the correct explanation is that the correlation is due to conditioning on a hidden common effect (namely the decision of whether to record the outcome or not, so we can censor tail-tail events). This is called **selection bias**.

As another example of this, consider a Gaussian DPGM of the form

$$p(x, y, z) = \mathcal{N}(x| -5, 1)\mathcal{N}(y|5, 1)\mathcal{N}(z|x + y, 1) \quad (4.25)$$

The graph structure is $X \rightarrow Z \leftarrow Y$, where Z is the child node. Some samples from the unconditional joint distribution $p(x, y, z)$ are shown in Figure 4.6(a); we see that X and Y are uncorrelated. Now suppose we only select samples where $z > 2.5$. Some samples from the conditional joint distribution

$p(x, y|z > 2.5)$ are shown in Figure 4.6(d); we see that now X and Y are correlated. This could cause us to erroneously conclude that there is a causal relationship, but in fact the dependency is caused by selection bias.

4.2.4.3 Markov blankets

The smallest set of nodes that renders a node i conditionally independent of all the other nodes in the graph is called i 's **Markov blanket**; we will denote this by $\text{mb}(i)$. Below we show that the Markov blanket of a node in a DPGM is equal to the parents, the children, and the **co-parents**, i.e., other nodes who are also parents of its children:

$$\text{mb}(i) \triangleq \text{ch}(i) \cup \text{pa}(i) \cup \text{cpa}(i) \quad (4.26)$$

See Figure 4.7 for an illustration.

To see why this is true, let us partition all the nodes into the target node X_i , its parents U , its children Y , its coparents Z , and the other variables O . Let X_{-i} be all the nodes except X_i . Then we have

$$p(X_i|X_{-i}) = \frac{p(X_i, X_{-i})}{\sum_x p(X_i = x, X_{-i})} \quad (4.27)$$

$$= \frac{p(X_i, U, Y, Z, O)}{\sum_x p(X_i = x, U, Y, Z, O)} \quad (4.28)$$

$$= \frac{p(X_i|U)[\prod_j p(Y_j|X_i, Z_j)]P(U, Z, O)}{\sum_x p(X_i = x|U)[\prod_j p(Y_j|X_i = x, Z_j)]P(U, Z, O)} \quad (4.29)$$

$$= \frac{p(X_i|U)[\prod_j p(Y_j|X_i, Z_j)]}{\sum_x p(X_i = x|U)[\prod_j p(Y_j|X_i = x, Z_j)]} \quad (4.30)$$

$$\propto p(X_i|\text{pa}(X_i)) \prod_{Y_j \in \text{ch}(X_i)} p(Y_j|\text{pa}(Y_j)) \quad (4.31)$$

where $\text{ch}(X_i)$ are the children of X_i and $\text{pa}(Y_j)$ are the parents of Y_j . We see that the terms that do not involve X_i cancel out from the numerator and denominator, so we are left with a product of terms that include X_i in their “scope”. Hence the **full conditional** for node i becomes

$$p(x_i|\mathbf{x}_{-i}) = p(x_i|\mathbf{x}_{\text{mb}(i)}) \propto p(x_i|\mathbf{x}_{\text{pa}(i)}) \prod_{k \in \text{ch}(i)} p(x_k|\mathbf{x}_{\text{pa}(k)}) \quad (4.32)$$

We will see applications of this in Gibbs sampling (Equation (12.19)), and mean field variational inference (Equation (10.87)).

4.2.4.4 Other Markov properties

From the d-separation criterion, one can conclude that

$$i \perp \text{nd}(i) \setminus \text{pa}(i) | \text{pa}(i) \quad (4.33)$$

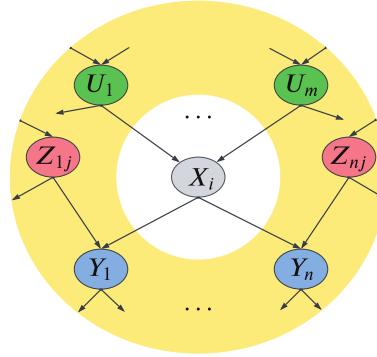


Figure 4.7: Illustration of the Markov blanket of a node in a directed graphical model. The target node X_i is shown in gray, its parents $U_{1:m}$ are shown in green, its children $Y_{1:n}$ are shown in blue, and its coparents $Z_{1:n,1:j}$ are shown in red. X_i is conditionally independent of all the other variables in the model given these variables. Adapted from Figure 13.4b of [RN19].

where the **non-descendants** of a node $\text{nd}(i)$ are all the nodes except for its descendants, $\text{nd}(i) = \{1, \dots, N_G\} \setminus \{i \cup \text{desc}(i)\}$. Equation (4.33) is called the (directed) **local Markov property**. For example, in Figure 4.23(a), we have $\text{nd}(3) = \{1, 2, 4\}$, and $\text{pa}(3) = 1$, so $3 \perp 2, 4 | 1$.

A special case of this property is when we only look at predecessors of a node according to some topological ordering. We have

$$i \perp \text{pred}(i) \setminus \text{pa}(i) | \text{pa}(i) \quad (4.34)$$

which follows since $\text{pred}(i) \subseteq \text{nd}(i)$. This is called the **ordered Markov property**, which justifies Equation (4.2). For example, in Figure 4.23(a), if we use the ordering $1, 2, \dots, 7$. we find $\text{pred}(3) = \{1, 2\}$ and $\text{pa}(3) = 1$, so $3 \perp 2 | 1$.

We have now described three Markov properties for DAGs: the directed global Markov property G in Equation (4.17), the directed local Markov property L in Equation (4.33), and the ordered Markov property O in Equation (4.34). It is obvious that $G \implies L \implies O$. What is less obvious, but nevertheless true, is that $O \implies L \implies G$ (see e.g., [KF09a] for the proof). Hence all these properties are equivalent.

Furthermore, any distribution p that is Markov wrt a graph can be factorized as in Equation (4.2); this is called the **factorization property** F . It is obvious that $O \implies F$, but one can show that the converse also holds (see e.g., [KF09a] for the proof).

4.2.5 Generation (sampling)

It is easy to generate prior samples from a DPGM: we simply visit the nodes in **topological order**, parents before children, and then sample a value for each node given the value of its parents. This will generate independent samples from the joint, $(x_1, \dots, x_{N_G}) \sim p(\mathbf{x}|\boldsymbol{\theta})$. This is called **ancestral sampling**.

4.2.6 Inference

In the context of PGMs, the term “**inference**” refers to the task of computing the posterior over a set of **query nodes** Q given the observed values for a set of **visible nodes** V , while marginalizing over the irrelevant **nuisance variables**, $R = \{1, \dots, N_G\} \setminus \{Q, V\}$:

$$p_{\theta}(Q|V) = \frac{p_{\theta}(Q, V)}{p_{\theta}(V)} = \frac{\sum_R p_{\theta}(Q, V, R)}{p_{\theta}(V)} \quad (4.35)$$

(If the variables are continuous, we should replace sums with integrals.) If Q is a single node, then $p_{\theta}(Q|V)$ is called the **posterior marginal** for node Q .

As an example, suppose $V = \mathbf{x}$ is a sequence of observed sound waves, $Q = \mathbf{z}$ is the corresponding set of unknown spoken words, and $R = \mathbf{r}$ are random “non-semantic” factors associated with the signal, such as prosody or background noise. Our goal is to compute the posterior over the words given the sounds, while being invariant to the irrelevant factors:

$$p_{\theta}(\mathbf{z}|\mathbf{x}) = \sum_{\mathbf{r}} p_{\theta}(\mathbf{z}, \mathbf{r}|\mathbf{x}) = \sum_{\mathbf{r}} \frac{p_{\theta}(\mathbf{z}, \mathbf{r}, \mathbf{x})}{p_{\theta}(\mathbf{x})} = \sum_{\mathbf{r}} \frac{p_{\theta}(\mathbf{z}, \mathbf{r}, \mathbf{x})}{\sum_{\mathbf{z}', \mathbf{r}'} p_{\theta}(\mathbf{z}', \mathbf{r}', \mathbf{x})} \quad (4.36)$$

As a simplification, we can “lump” the random factors R into the query set Q to define the complete set of **hidden variables** $H = Q \cup R$. In this case, the tasks simplifies to

$$p_{\theta}(\mathbf{h}|\mathbf{x}) = \frac{p_{\theta}(\mathbf{h}, \mathbf{x})}{p_{\theta}(\mathbf{x})} = \frac{p_{\theta}(\mathbf{h}, \mathbf{x})}{\sum_{\mathbf{h}'} p_{\theta}(\mathbf{h}', \mathbf{x})} \quad (4.37)$$

The computational complexity of the inference task depends on the CI properties of the graph, as we discuss in Chapter 9. In general it is NP-hard (see Section 9.5.4), but for certain graph structures (such as chains, trees, and other sparse graphs), it can be solved efficiently (in polynomial) time using dynamic programming (see Chapter 9). For cases where it is intractable, we can use standard methods for approximate Bayesian inference, which we review in Chapter 7.

4.2.6.1 Example: inference in the student network

As an example of inference in PGMs, consider the student network from Section 4.2.2.2. Suppose we observe that the student gets a grade of C. The posterior marginals are shown in Figure 4.8a. We see that the low grade could be explained by the class being hard (since $p(D = \text{Hard}|G = C) = 0.63$), but is more likely explained by the student having low intelligence (since $p(I = \text{High}|G = C) = 0.08$).

However, now suppose we *also* observe that the student gets a good SAT score. The new posterior marginals are shown in Figure 4.8b. Now the posterior probability that the student is intelligent has jumped to $p(I = \text{High}|G = C, \text{SAT} = \text{Good}) = 0.58$, since otherwise it would be difficult to explain the good SAT score. Once we believe the student has high intelligence, we have to explain the C grade by assuming the class is hard, and indeed we find that the probability that the class is hard has increased to $p(D = \text{Hard}|G = C) = 0.76$. (This negative mutual interaction between multiple causes of some observations is called the explaining away effect, and is discussed in Section 4.2.4.2.)

4.2.7 Learning

So far, we have assumed that the structure G and parameters θ of the PGM are known. However, it is possible to learn both of these from data. For details on how to learn G from data, see Section 30.3.

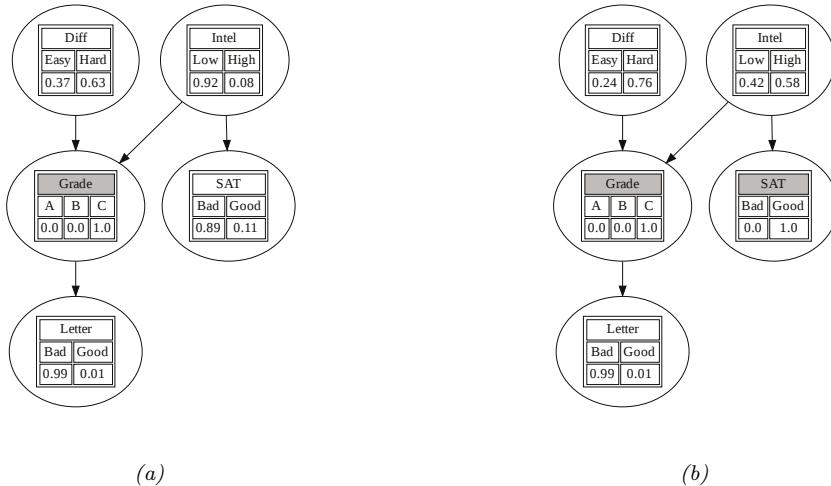


Figure 4.8: Illustration of belief updating in the “Student” PGM. The histograms show the marginal distribution of each node. Nodes with shaded titles are clamped to an observed value. (a) Posterior after conditioning on Grade=C. (b) Posterior after also conditioning on SAT=Good. Generated by `student_pgm.ipynb`.

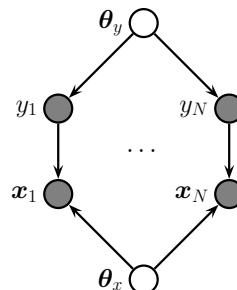


Figure 4.9: A DPGM representing the joint distribution $p(y_{1:N}, x_{1:N}, \theta_y, \theta_x)$. Here θ_x and θ_y are global parameter nodes that are shared across the examples, whereas x_n and y_n are local variables.

Here we focus on **parameter learning**, i.e., computing the posterior $p(\boldsymbol{\theta}|\mathcal{D}, G)$. (Henceforth we will drop the conditioning on G , since we assume the graph structure is fixed.)

We can compute the parameter posterior $p(\theta|\mathcal{D})$ by treating θ as “just another hidden variable”, and then performing inference. However, in the machine learning community, it is more common to just compute a point estimate of the parameters, such as the posterior mode, $\hat{\theta} = \text{argmax } p(\theta|\mathcal{D})$. This approximation is often reasonable, since the parameters depend on all the data, rather than just a single datapoint, and are therefore less uncertain than other hidden variables.

4.2.7.1 Learning from complete data

Figure 4.9 represents a graphical model for a typical supervised learning problem. We have N **local variables**, \mathbf{x}_n and y_n , and 2 **global variables**, corresponding to the parameters, which are shared across data samples. The local variables are observed (in the training set), so they are represented by solid (shaded) nodes. The global variables are not observed, and hence are represented by empty (unshaded) nodes. (The model represents a generative classifier, so the edge is from y_n to \mathbf{x}_n ; if we are fitting a discriminative classifier, the edge would be from \mathbf{x}_n to y_n , and there would be no θ_y prior node.)

From the CI properties of Figure 4.9, it follows that the joint distribution factorizes into a product of terms, one per node:

$$p(\boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\theta}_x)p(\boldsymbol{\theta}_y) \left[\prod_{n=1}^N p(y_n | \boldsymbol{\theta}_y) p(\mathbf{x}_n | y_n, \boldsymbol{\theta}_x) \right] \quad (4.38)$$

$$= \left[p(\boldsymbol{\theta}_y) \prod_{n=1}^N p(y_n | \boldsymbol{\theta}_y) \right] \left[p(\boldsymbol{\theta}_x) \prod_{n=1}^N p(\mathbf{x}_n | y_n, \boldsymbol{\theta}_x) \right] \quad (4.39)$$

$$= [p(\boldsymbol{\theta}_y)p(\mathcal{D}_y | \boldsymbol{\theta}_y)] [p(\boldsymbol{\theta}_x)p(\mathcal{D}_x | \boldsymbol{\theta}_x)] \quad (4.40)$$

where $\mathcal{D}_y = \{y_n\}_{n=1}^N$ is the data that is sufficient for estimating $\boldsymbol{\theta}_y$ and $\mathcal{D}_x = \{\mathbf{x}_n, y_n\}_{n=1}^N$ is the data that is sufficient for $\boldsymbol{\theta}_x$.

From Equation (4.40), we see that the prior, likelihood, and posterior all **decompose** or factorize according to the graph structure. Thus we can compute the posterior for each parameter independently. In general, we have

$$p(\boldsymbol{\theta}, \mathcal{D}) = \prod_{i=1}^{N_G} p(\boldsymbol{\theta}_i)p(\mathcal{D}_i | \boldsymbol{\theta}_i) \quad (4.41)$$

Hence the likelihood and prior factorizes, and thus so does the posterior. If we just want to compute the MLE, we can compute

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N_G} p(\mathcal{D}_i | \boldsymbol{\theta}_i) \quad (4.42)$$

We can solve this for each node independently, as we illustrate in Section 4.2.7.2.

4.2.7.2 Example: computing the MLE for CPTs

In this section, we illustrate how to compute the MLE for tabular CPDs. The likelihood is given by the following product of multinomials:

$$p(\mathcal{D} | \boldsymbol{\theta}) = \prod_{n=1}^N \prod_{i=1}^{N_G} p(x_{ni} | \mathbf{x}_{n,\text{pa}(i)}, \boldsymbol{\theta}_i) \quad (4.43)$$

$$= \prod_{n=1}^N \prod_{i=1}^{N_G} \prod_{j=1}^{J_i} \prod_{k=1}^{K_i} \theta_{ijk}^{\mathbb{I}(x_{ni}=k, \mathbf{x}_{n,\text{pa}(i)}=j)} \quad (4.44)$$

I	D	G	S	L
0	0	2	0	0
0	1	2	0	0
0	0	1	1	1
1	1	1	1	0
1	0	0	1	1
0	0	0	0	1
1	1	2	1	1

Table 4.2: Some fully observed training data for the student network.

I	D	$N_{i,j,k}$	$\hat{\theta}_{i,j,k}$	$\bar{\theta}_{i,j,k}$
0	0	[1, 1, 1]	[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]	[\frac{2}{6}, \frac{2}{6}, \frac{2}{6}]
0	1	[0, 0, 1]	[\frac{0}{1}, \frac{0}{1}, \frac{1}{1}]	[\frac{1}{4}, \frac{1}{4}, \frac{2}{4}]
1	0	[1, 0, 0]	[\frac{1}{1}, \frac{0}{1}, \frac{0}{1}]	[\frac{2}{4}, \frac{1}{4}, \frac{1}{4}]
1	1	[0, 1, 1]	[\frac{0}{2}, \frac{1}{2}, \frac{1}{2}]	[\frac{1}{5}, \frac{2}{5}, \frac{2}{5}]

Table 4.3: Sufficient statistics N_{ijk} and corresponding MLE $\hat{\theta}_{ijk}$ and posterior mean $\bar{\theta}_{ijk}$ (with Dirichlet (1,1,1) prior) for node $i = G$ in the student network. Each row corresponds to a different joint configuration of its parent nodes, corresponding to state j . The index k refers to the 3 possible values of the child node G .

where

$$\theta_{ijk} \triangleq p(x_i = k | \mathbf{x}_{\text{pa}(i)} = j) \quad (4.45)$$

Let us define the sufficient statistics for node i to be N_{ijk} , which is the number of times that node i is in state k while its parents are in joint state j :

$$N_{ijk} \triangleq \sum_{n=1}^N \mathbb{I}(x_{n,i} = k, x_{n,\text{pa}(i)} = j) \quad (4.46)$$

The MLE for a multinomial is given by the normalized empirical frequencies:

$$\hat{\theta}_{ijk} = \frac{N_{ijk}}{\sum_{k'} N_{ijk'}} \quad (4.47)$$

For example, consider the student network from Section 4.2.2.2. In Table 4.2, we show some sample training data. For example, the last line in the tabel encodes a student who is smart ($I = 1$), who takes a hard class ($D = 1$), gets a C ($G = 2$), but who does well on the SAT ($S = 1$) and gets a good letter of recommendation ($L = 1$).

In Table 4.3, we list the sufficient statistics N_{ijk} and the MLE $\hat{\theta}_{ijk}$ for node $i = G$, with parents (I, D) . A similar process can be used for the other nodes. Thus we see that fitting a DPGM with tabular CPDs reduces to a simple counting problem.

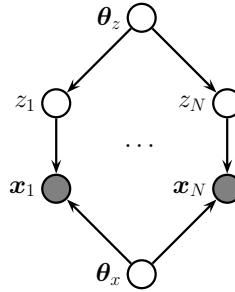


Figure 4.10: A DPGM representing the joint distribution $p(z_{1:N}, \mathbf{x}_{1:N}, \boldsymbol{\theta}_z, \boldsymbol{\theta}_x)$. The local variables \mathbf{z}_n are hidden, whereas \mathbf{x}_n are observed. This is typical for learning unsupervised latent variable models.

However, we notice there are a lot of zeros in the sufficient statistics, due to the small sample size, resulting in extreme estimates for some of the probabilities $\hat{\theta}_{ijk}$. We discuss a (Bayesian) solution to this in Section 4.2.7.3.

4.2.7.3 Example: computing the posterior for CPTs

In Section 4.2.7.2 we discussed how to compute the MLE for the CPTs in a discrete Bayes net. We also observed that this can suffer from the zero-count problem. In this section, we show how a Bayesian approach can solve this problem.

Let us put a separate Dirichlet prior on each row of each CPT, i.e., $\boldsymbol{\theta}_{ij} \sim \text{Dir}(\boldsymbol{\alpha}_{ij})$. Then we can compute the posterior by simply adding the pseudocounts to the empirical counts to get $\boldsymbol{\theta}_{ij}|\mathcal{D} \sim \text{Dir}(\mathbf{N}_{ij} + \boldsymbol{\alpha}_{ij})$, where $\mathbf{N}_{ij} = \{N_{ijk} : k = 1 : K_i\}$, and N_{ijk} is the number of times that node i is in state k while its parents are in state j . Hence the posterior mean estimate is given by

$$\bar{\theta}_{ijk} = \frac{N_{ijk} + \alpha_{ijk}}{\sum_{k'} (N_{ijk'} + \alpha_{ijk'})} \quad (4.48)$$

The MAP estimate has the same form, except we use $\alpha_{ijk} - 1$ instead of α_{ijk} .

In Table 4.3, we illustrate this approach applied to the G node in the student network, where we use a uniform Dirichlet prior, $\alpha_{ijk} = 1$.

4.2.7.4 Learning from incomplete data

In Section 4.2.7.1, we explained that when we have complete data, the likelihood (and posterior) factorizes over CPDs, so we can estimate each CPD independently. Unfortunately, this is no longer the case when we have incomplete or missing data. To see this, consider Figure 4.10. The likelihood of the observed data can be written as follows:

$$p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{\mathbf{z}_{1:N}} \left[\prod_{n=1}^N p(\mathbf{z}_n|\boldsymbol{\theta}_z) p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}_x) \right] \quad (4.49)$$

$$= \prod_{n=1}^N \sum_{\mathbf{z}_n} p(\mathbf{z}_n|\boldsymbol{\theta}_z) p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}_x) \quad (4.50)$$

Thus the log likelihood is given by

$$\ell(\boldsymbol{\theta}) = \sum_n \log \sum_{\mathbf{z}_n} p(\mathbf{z}_n | \boldsymbol{\theta}_z) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x) \quad (4.51)$$

The log function does not distribute over the $\sum_{\mathbf{z}_n}$ operation, so the objective does not decompose over nodes.² Consequently, we can no longer compute the MLE or the posterior by solving separate problems per node.

To solve this, we will resort to optimization methods. (We focus on the MLE case, and leave discussion of Bayesian inference for latent variable models to Part II.) In the sections below, we discuss how to use EM and SGD to find a local optimum of the (non convex) log likelihood objective.

4.2.7.5 Using EM to fit CPTs in the incomplete data case

A popular method for estimating the parameters of a DPGM in the presence of missing data is to use the expectation maximization (EM) algorithm, as proposed in [Lau95]. We describe EM in detail in Section 6.5.3, but the basic idea is to alternate between inferring the latent variables \mathbf{z}_n (the E or expectation step), and estimating the parameters given this completed dataset (the M or maximization step). Rather than returning the full posterior $p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta}^{(t)})$ in the E step, we instead return the expected sufficient statistics (ESS), which takes much less space. In the M step, we maximize the expected value of the log likelihood of the fully observed data using these ESS.

As an example, suppose all the CPDs are tabular, as in the example in Section 4.2.7.2. The log-likelihood of the complete data is given by

$$\log p(\mathcal{D} | \boldsymbol{\theta}) = \sum_{i=1}^{N_G} \sum_{j=1}^{J_i} \sum_{k=1}^{K_i} N_{ijk} \log \theta_{ijk} \quad (4.52)$$

and hence the expected complete data log-likelihood has the form

$$\mathbb{E} [\log p(\mathcal{D} | \boldsymbol{\theta})] = \sum_i \sum_j \sum_k \bar{N}_{ijk} \log \theta_{ijk} \quad (4.53)$$

where

$$\bar{N}_{ijk} = \sum_{n=1}^N \mathbb{E} [\mathbb{I}(x_{ni} = k, \mathbf{x}_{n,\text{pa}(i)} = j)] = \sum_{n=1}^N p(x_{ni} = k, \mathbf{x}_{n,\text{pa}(i)} = j | \mathcal{D}_n, \boldsymbol{\theta}^{old}) \quad (4.54)$$

where \mathcal{D}_n are all the visible variables in case n , and $\boldsymbol{\theta}^{old}$ are the parameters from the previous iteration. The quantity $p(x_{ni}, \mathbf{x}_{n,\text{pa}(i)} | \mathcal{D}_n, \boldsymbol{\theta}^{old})$ is known as a **family marginal**, and can be computed using any GM inference algorithm. The \bar{N}_{ijk} are the **expected sufficient statistics** (ESS), and constitute the output of the E step.

2. We can also see this from the graphical model: $\boldsymbol{\theta}_x$ is no longer independent of $\boldsymbol{\theta}_z$, because there is a path that connects them via the hidden nodes \mathbf{z}_n . (See Section 4.2.4 for an explanation of how to “read off” such CI properties from a DPGM.)

Given these ESS, the M step has the simple form

$$\hat{\theta}_{ijk} = \frac{\bar{N}_{ijk}}{\sum_{k'} \bar{N}_{ijk'}} \quad (4.55)$$

We can modify this to perform MAP estimation with a Dirichlet prior by simply adding pseudocounts to the expected counts.

The famous Baum-Welch algorithm is a special case of the above equations which arises when the DPGM is an HMM (see Section 29.4.1 for details).

4.2.7.6 Using SGD to fit CPTs in the incomplete data case

The EM algorithm is a batch algorithm. To scale up to large datasets, it is more common to use stochastic gradient descent or SGD (see e.g., [BC94; Bin+97]). To apply this, we need to compute the marginal likelihood of the observed data for each example:

$$p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{\mathbf{z}_n} p(\mathbf{z}_n | \boldsymbol{\theta}_z) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x) \quad (4.56)$$

where $\boldsymbol{\theta} = (\boldsymbol{\theta}_z, \boldsymbol{\theta}_x)$. (We say that we have “collapsed” the model by marginalizing out \mathbf{z}_n .) We can then compute the log likelihood using

$$\ell(\boldsymbol{\theta}) = \log p(\mathcal{D} | \boldsymbol{\theta}) = \log \prod_{n=1}^N p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{n=1}^N \log p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (4.57)$$

The gradient of this objective can be computed as follows:

$$\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \sum_n \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (4.58)$$

$$= \sum_n \frac{1}{p(\mathbf{x}_n | \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (4.59)$$

$$= \sum_n \frac{1}{p(\mathbf{x}_n | \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \left[\sum_{\mathbf{z}_n} p(\mathbf{z}_n, \mathbf{x}_n | \boldsymbol{\theta}) \right] \quad (4.60)$$

$$= \sum_n \sum_{\mathbf{z}_n} \frac{p(\mathbf{z}_n, \mathbf{x}_n | \boldsymbol{\theta})}{p(\mathbf{x}_n | \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{z}_n, \mathbf{x}_n | \boldsymbol{\theta}) \quad (4.61)$$

$$= \sum_n \sum_{\mathbf{z}_n} p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{z}_n, \mathbf{x}_n | \boldsymbol{\theta}) \quad (4.62)$$

We can now apply a minibatch approximation to this in the usual way.

4.2.8 Plate notation

To make the parameters of a PGM explicit, we can add them as nodes to the graph, and treat them as hidden variables to be inferred. Figure 4.11(a) shows a simple example, in which we have N iid

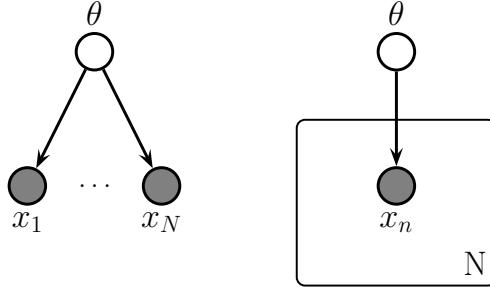


Figure 4.11: Left: datapoints \mathbf{x}_n are conditionally independent given $\boldsymbol{\theta}$. Right: Same model, using plate notation. This represents the same model as the one on the left, except the repeated \mathbf{x}_n nodes are inside a box, known as a plate; the number in the lower right hand corner, N , specifies the number of repetitions of the \mathbf{x}_n node.

random variables, \mathbf{x}_n , all drawn from the same distribution with common parameter $\boldsymbol{\theta}$. We denote this by

$$\mathbf{x}_n \sim p(\mathbf{x}|\boldsymbol{\theta}) \quad (4.63)$$

The corresponding joint distribution over the parameters and data $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ has the form

$$p(\mathcal{D}, \boldsymbol{\theta}) = p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta}) \quad (4.64)$$

where $p(\boldsymbol{\theta})$ is the prior distribution for the parameters, and $p(\mathcal{D}|\boldsymbol{\theta})$ is the likelihood. By virtue of the iid assumption, the likelihood can be rewritten as follows:

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^N p(\mathbf{x}_n|\boldsymbol{\theta}) \quad (4.65)$$

Notice that the order of the data vectors is not important for defining this model, i.e., we can permute the leaves of the DPGM. When this property holds, we say that the data is **exchangeable**.

In Figure 4.11(a), we see that the \mathbf{x} nodes are repeated N times. (The shaded nodes represent observed values, whereas the unshaded (hollow) nodes represent latent variables or parameters.) To avoid visual clutter, it is common to use a form of **syntactic sugar** called **plates**. This is a notational convention in which we draw a little box around the repeated variables, with the understanding that nodes within the box will get repeated when the model is **unrolled**. We often write the number of copies or repetitions in the bottom right corner of the box. This is illustrated in Figure 4.11(b).

4.2.8.1 Example: factor analysis

In Section 28.3.1, we discuss the factor analysis model, which has the form

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \quad (4.66)$$

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Psi}) \quad (4.67)$$

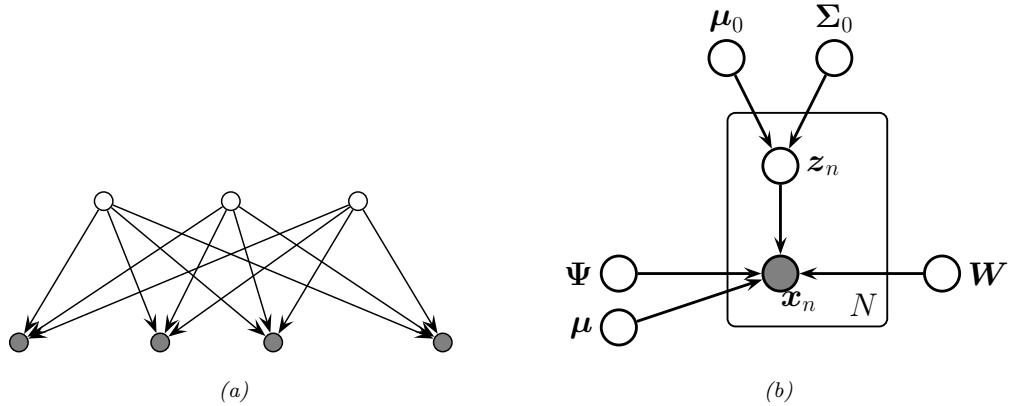


Figure 4.12: (a) Factor analysis model illustrated as a DPGM. We show the components of \mathbf{z} (top row) and \mathbf{x} (bottom row) as individual scalar nodes. (b) Equivalent model, where \mathbf{z} and \mathbf{x} are collapsed to vector-valued nodes, and parameters are added, using plate notation.

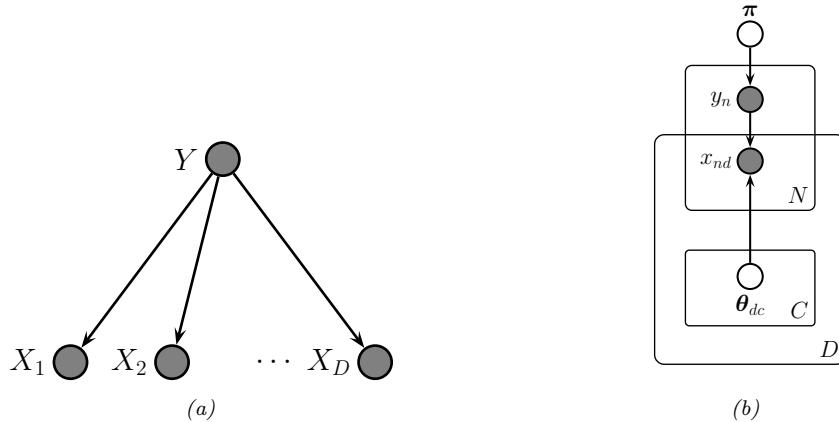


Figure 4.13: (a) Naive Bayes classifier as a DPGM. (b) Model augmented with plate notation.

where \mathbf{W} is a $D \times L$ matrix, known as the factor loading matrix, and Ψ is a diagonal $D \times D$ covariance matrix.

Note that \mathbf{z} and \mathbf{x} are both vectors. We can explicitly represent their components as scalar nodes as in Figure 4.12a. Here the directed edges correspond to non-zero entries in the \mathbf{W} matrix.

We can also explicitly show the parameters of the model, using plate notation, as shown in Figure 4.12b.

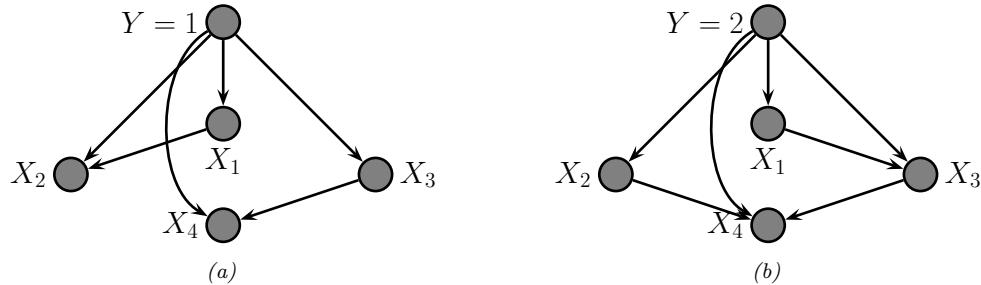


Figure 4.14: Tree-augmented naive Bayes classifier for $D = 4$ features. The tree topology can change depending on the value of y , as illustrated.

4.2.8.2 Example: naive Bayes classifier

In some models, we have doubly indexed variables. For example, consider a **naive Bayes classifier**. This is a simple generative classifier, defined as follows:

$$p(\mathbf{x}, y|\boldsymbol{\theta}) = p(y|\boldsymbol{\pi}) \prod_{d=1}^D p(x_d|y, \boldsymbol{\theta}_d) \quad (4.68)$$

The fact that the features $\mathbf{x}_{1:D}$ are considered conditionally independent given the class label y is where the term “naive” comes from. Nevertheless, this model often works surprisingly well, and is extremely easy to fit.

We can represent the conditional independence assumption as shown in Figure 4.13a. We can represent the repetition over the dimension d with a plate. When we turn to inferring the parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, \boldsymbol{\theta}_{1:D, 1:C})$, we also need to represent the repetition over data cases n . This is shown in Figure 4.13b. Note that the parameter $\boldsymbol{\theta}_{dc}$ depends on d and c , whereas the feature \mathbf{x}_{nd} depends on n and d . This is shown using **nested plates** to represent the shared d index.

4.2.8.3 Example: relaxing the naive Bayes assumption

We see from Figure 4.13a that the observed features are conditionally independent given the class label. We can of course allow for dependencies between the features, as illustrated in Figure 4.14. (We omit parameter nodes for simplicity.) If we enforce that the edges between the features forms a tree the model is known as a **tree-augmented naive Bayes classifier** [FGG97], or **TAN** model. (Trees are a restricted form of graphical model that have various computational advantages that we discuss later.) Note that the topology of the tree can change depending on the value of the class node y ; in this case, the model is known as a **Bayesian multi net**, and can be thought of as a supervised mixture of trees.

4.3 Undirected graphical models (Markov random fields)

Directed graphical models (Section 4.2) are very useful. However, for some domains, being forced to choose a direction for the edges, as required by a DAG, is rather awkward. For example, consider

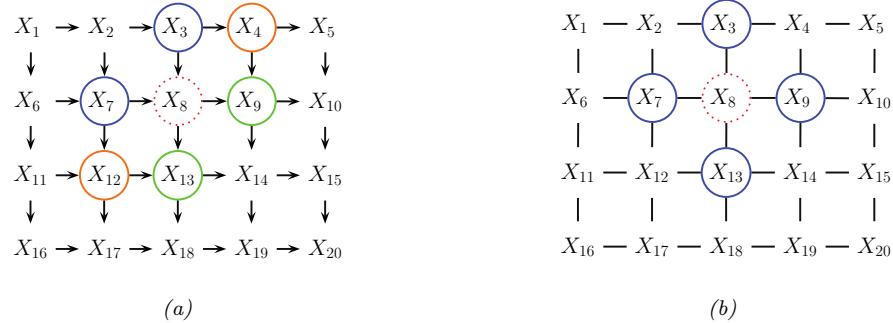


Figure 4.15: (a) A 2d lattice represented as a DAG. The dotted red node X_8 is independent of all other nodes (black) given its Markov blanket, which include its parents (blue), children (green) and co-parents (orange). (b) The same model represented as a UPGM. The red node X_8 is independent of the other black nodes given its neighbors (blue nodes).

modeling an image. It is reasonable to assume that the intensity values of neighboring pixels are correlated. We can model this using a DAG with a 2d lattice topology as shown in Figure 4.15(a). This is known as a **Markov mesh** [AHK65]. However, its conditional independence properties are rather unnatural.

An alternative is to use an undirected probabilistic graphical model (**UPGM**), also called a **Markov random field (MRF)** or **Markov network**. These do not require us to specify edge orientations, and are much more natural for some problems such as image analysis and spatial statistics. For example, an undirected 2d lattice is shown in Figure 4.15(b); now the Markov blanket of each node is just its nearest neighbors, as we show in Section 4.3.6.

Roughly speaking, the main advantages of UPGMs over DPGMs are: (1) they are symmetric and therefore more “natural” for certain domains, such as spatial or relational data; and (2) discriminative UPGMs (aka conditional random fields, or CRFs), which define conditional densities of the form $p(\mathbf{y}|\mathbf{x})$, work better than discriminative DGMs, for reasons we explain in Section 4.5.3. The main disadvantages of UPGMs compared to DPGMs are: (1) the parameters are less interpretable and less modular, for reasons we explain in Section 4.3.1; and (2) it is more computationally expensive to estimate the parameters, for reasons we explain in Section 4.3.9.1.

4.3.1 Representing the joint distribution

Since there is no topological ordering associated with an undirected graph, we can't use the chain rule to represent $p(\mathbf{x}_{1:N_G})$. So instead of associating CPDs with each node, we associate **potential functions** or **factors** with each **maximal clique** in the graph.³ We will denote the potential function for clique c by $\psi_c(\mathbf{x}_c; \theta_c)$, where θ_c are its parameters. A potential function can be any non-negative function of its arguments (we give some examples below). We can use these functions to define the joint distribution as we explain in Section 4.3.1.1.

3. A **clique** is a set of nodes that are all neighbors of each other. A **maximal clique** is a clique which cannot be made any larger without losing the clique property.

4.3.1.1 Hammersley-Clifford theorem

Suppose a joint distribution p satisfies the CI properties implied by the undirected graph G . (We discuss how to derive these properties in Section 4.3.6.) Then the **Hammersley-Clifford theorem** tells us that p can be written as follows:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) \quad (4.69)$$

where \mathcal{C} is the set of all the (maximal) cliques of the graph G , and $Z(\boldsymbol{\theta})$ is the **partition function** given by

$$Z(\boldsymbol{\theta}) \triangleq \sum_{\mathbf{x}} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) \quad (4.70)$$

Note that the partition function is what ensures the overall distribution sums to 1.⁴

The Hammersley-Clifford theorem was never published, but a proof can be found in [KF09a]. (Note that the theorem only holds for positive distributions, i.e., ones where $p(\mathbf{x}|\boldsymbol{\theta}) > 0$ for all configurations \mathbf{x} , which rules out some models with hard constraints.)

4.3.1.2 Gibbs distribution

The distribution in Equation (4.69) can be rewritten as follows:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-\mathcal{E}(\mathbf{x}; \boldsymbol{\theta})) \quad (4.71)$$

where $\mathcal{E}(\mathbf{x}) > 0$ is the **energy** of state \mathbf{x} , defined by

$$\mathcal{E}(\mathbf{x}; \boldsymbol{\theta}) = \sum_c \mathcal{E}(\mathbf{x}_c; \boldsymbol{\theta}_c) \quad (4.72)$$

where \mathbf{x}_c are the variables in clique c . We can see the equivalence by defining the clique potentials as

$$\psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) = \exp(-\mathcal{E}(\mathbf{x}_c; \boldsymbol{\theta}_c)) \quad (4.73)$$

We see that low energy is associated with high probability states.

Equation (4.71) is known as the **Gibbs distribution**. This kind of probability model is also called an **energy-based model**. These are commonly used in physics and biochemistry. They are also used in ML to define generative models, as we discuss in Chapter 24. (See also Section 4.4, where we discuss conditional random fields (CRFs), which are models of the form $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$, where the potential functions are conditioned on input features \mathbf{x} .)

4.3.2 Fully visible MRFs (Ising, Potts, Hopfield, etc.)

In this section, we discuss some UPGMs for 2d grids, that are used in statistical physics and computer vision. We then discuss extensions to other graph structures, which are useful for biological modeling and pattern completion.

4. The partition function is denoted by Z because of the German word *Zustandssumme*, which means “sum over states”. This reflects the fact that a lot of pioneering working on MRFs was done by German (and Austrian) physicists, such as Boltzmann.

4.3.2.1 Ising models

Consider the 2d lattice in Figure 4.15(b). We can represent the joint distribution as follows:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{i \sim j} \psi_{ij}(x_i, x_j; \boldsymbol{\theta}) \quad (4.74)$$

where $i \sim j$ means i and j are neighbors in the graph. This is called a **2d lattice model**.

An **Ising model** is a special case of the above, where the variables x_i are binary. Such models are often used to represent magnetic materials. In particular, each node represents an atom, which can have a magnetic dipole, or **spin**, which is in one of two states, +1 and -1. In some magnetic systems, neighboring spins like to be similar; in other systems, they like to be dissimilar. We can capture this interaction by defining the clique potentials as follows:

$$\psi_{ij}(x_i, x_j; \boldsymbol{\theta}) = \begin{cases} e^{J_{ij}} & \text{if } x_i = x_j \\ e^{-J_{ij}} & \text{if } x_i \neq x_j \end{cases} \quad (4.75)$$

where J_{ij} is the coupling strength between nodes i and j . This is known as the **Ising model**. If two nodes are not connected in the graph, we set $J_{ij} = 0$. We assume that the weight matrix is symmetric, so $J_{ij} = J_{ji}$. Often we also assume all edges have the same strength, so $J_{ij} = J$ for each (i, j) edge. Thus

$$\psi_{ij}(x_i, x_j; J) = \begin{cases} e^J & \text{if } x_i = x_j \\ e^{-J} & \text{if } x_i \neq x_j \end{cases} \quad (4.76)$$

It is more common to define the Ising model as an energy-based model, as follows:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(J)} \exp(-\mathcal{E}(\mathbf{x}; J)) \quad (4.77)$$

$$\mathcal{E}(\mathbf{x}; J) = -J \sum_{i \sim j} x_i x_j \quad (4.78)$$

where $\mathcal{E}(\mathbf{x}; J)$ is the energy, and where we exploited the fact that $x_i x_j = -1$ if $x_i \neq x_j$, and $x_i x_j = +1$ if $x_i = x_j$. The magnitude of J controls the degree of coupling strength between neighboring sites, which depends on the (inverse) temperature of the system (colder = more tightly coupled = larger magnitude J).

If all the edge weights are positive, $J > 0$, then neighboring spins are likely to be in the same state, since if $x_i = x_j$, the energy term gets a contribution of $-J < 0$, and lower energy corresponds to higher probability. In the machine learning literature, this is called an **associative Markov network**. In the physics literature, this is called a **ferromagnetic** model. If the weights are sufficiently strong, the corresponding probability distribution will have two modes, corresponding to the two checkerboard patterns in Figure 4.16a. These are called the **ground states** of the system.

If all of the weights are negative, $J < 0$, then the spins want to be different from their neighbors (see Figure 4.16b). This is called an **antiferromagnetic** system, and results in a **frustrated system**, since it is not possible for all neighbors to be different from each other in a 2d lattice. Thus the corresponding probability distribution will have multiple modes, corresponding to different “solutions” to the problem.

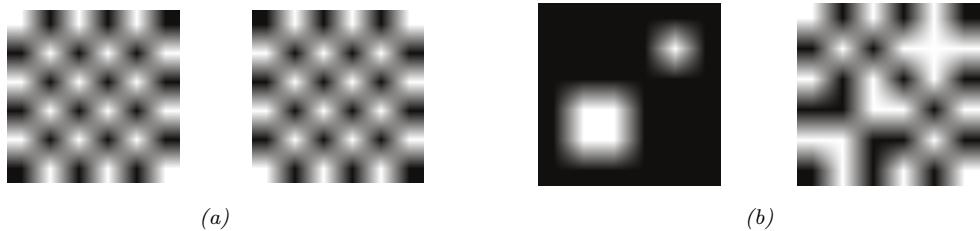


Figure 4.16: (a) The two ground states for a small ferromagnetic Ising model where $J = 1$. (b) Two different states for a small Ising model which have the same energy. Left: $J = 1$, so neighboring pixels have similar values. Right: $J = -1$, so neighboring pixels have different values. From Figures 31.7 and 31.8 of [Mac03].

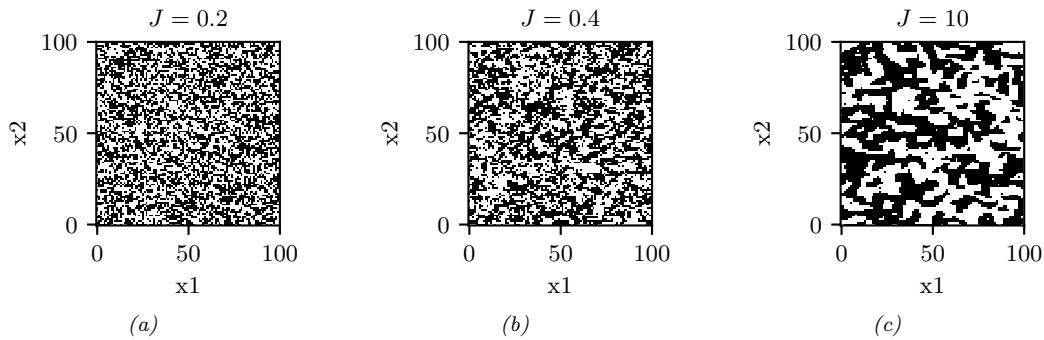


Figure 4.17: Samples from an associative Ising model with varying $J > 0$. Generated by `gibbs_demo_ising.ipynb`.

Figure 4.17 shows some samples from the Ising model for varying $J > 0$. (The samples were created using the Gibbs sampling method discussed in Section 12.3.3.) As the temperature reduces, the distribution becomes less entropic, and the “clumpiness” of the samples increases. One can show that, as the lattice size goes to infinity, there is a **critical temperature** J_c below which many large clusters occur, and above which many small clusters occur. In the case of an isotropic square lattice model, one can show [Geo88] that

$$J_c = \frac{1}{2} \log(1 + \sqrt{2}) \approx 0.44 \quad (4.79)$$

This rapid change in global behavior as we vary a parameter of the system is called a **phase transition**. This can be used to explain how natural systems, such as water, can suddenly go from solid to liquid, or from liquid to gas, when the temperature changes slightly. See e.g., [Mac03, ch 31] for further details on the statistical mechanics of Ising models.

In addition to pairwise terms, it is standard to add **unary terms**, $\psi_i(x_i)$. In statistical physics, this is called an **external field**. The resulting model is as follows:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_i \psi_i(x_i; \boldsymbol{\theta}) \prod_{i \sim j} \psi_{ij}(x_i, x_j; \boldsymbol{\theta}) \quad (4.80)$$

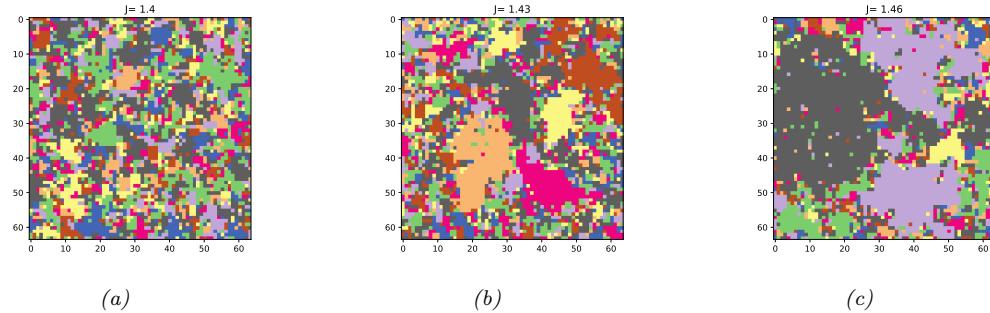


Figure 4.18: Visualizing a sample from a 10-state Potts model of size 128×128 . The critical value is $J_c = \log(1 + \sqrt{10}) = 1.426$. for different association strengths: (a) $J = 1.40$, (b) $J = 1.43$, (c) $J = 1.46$. Generated by [gibbs_demo_potts.ipynb](#).

The ψ_i terms can be thought of as a local bias term that is independent of the contributions of the neighboring nodes. For binary nodes, we can define this as follows:

$$\psi_i(x_i) = \begin{cases} e^\alpha & \text{if } x_i = +1 \\ e^{-\alpha} & \text{if } x_i = -1 \end{cases} \quad (4.81)$$

If we write this as an energy-based model, we have

$$\mathcal{E}(\mathbf{x}|\boldsymbol{\theta}) = -\alpha \sum_i x_i - J \sum_{i \sim j} x_i x_j \quad (4.82)$$

4.3.2.2 Potts models

In Section 4.3.2.1, we discussed the Ising model, which is a simple 2d MRF for defining distributions over binary variables. It is easy to generalize the Ising model to multiple discrete states, $x_i \in \{1, 2, \dots, K\}$. If we use the same potential function for every edge, we can write

$$\psi_{ij}(x_i = k, x_j = k') = e^{J_{ij}(k, k')} \quad (4.83)$$

where $J_{ij}(k, k')$ is the energy if one node has state k and its neighbor has state k' . A common special case is

$$\psi_{ij}(x_i = k, x_j = k') = \begin{cases} e^J & \text{if } k = k' \\ e^0 & \text{if } k \neq k' \end{cases} \quad (4.84)$$

This is called the **Potts model**. The Potts model reduces to the Ising model if we define $J_{\text{potts}} = 2J_{\text{ising}}$.

If $J > 0$, then neighboring nodes are encouraged to have the same label; this is an example of an associative Markov model. Some samples from this model are shown in Figure 4.18. The phase transition for a 2d Potts model occurs at the following value (see [MS96]):

$$J_c = \log(1 + \sqrt{K}) \quad (4.85)$$

We can extend this model to have local evidence for each node. If we write this as an energy-based model, we have

$$\mathcal{E}(\mathbf{x}|\boldsymbol{\theta}) = -\sum_i \sum_{k=1}^K \alpha_k \mathbb{I}(x_i = k) - J \sum_{i \sim j} \mathbb{I}(x_i = x_j) \quad (4.86)$$

4.3.2.3 Potts models for protein structure prediction

One interesting application of Potts models arises in the area of **protein structure prediction**. The goal is to predict the 3d shape of a protein from its 1d sequence of amino acids. A common approach to this is known as **direct coupling analysis** (DCA). We give a brief summary below; for details, see [Mor+11].

First we compute a **multiple sequence alignment** (MSA) from a set of related amino acid sequences from the same protein family; this can be done using HMMs, as explained in Section 29.3.2. The MSA can be represented by an $N \times T$ matrix \mathbf{X} , where N is the number of sequences, T is the length of each sequence, and $X_{ni} \in \{1, \dots, V\}$ is the identity of the letter at location i in sequence n . For protein sequences, $V = 21$, representing the 20 amino acids plus the gap character.

Once we have the MSA matrix \mathbf{X} , we fit the Potts model using maximum likelihood estimation, or some approximation, such as pseudolikelihood [Eke+13]; see Section 4.3.9 for details.⁵ After fitting the model, we select the edges with the highest J_{ij} coefficients, where $i, j \in \{1, \dots, T\}$ are locations or **residues** in the protein. Since these locations are highly coupled, they are likely to be in physical contact, since interacting residues must coevolve to avoid destroying the function of the protein (see e.g., [LHF17] for a review). This graph is called a **contact map**.

Once the contact map is established, it can be used as input to a 3d structural prediction algorithm, such as [Xu18] or the **alphafold** system [Eva+18], which won the 2018 CASP competition. Such methods use neural networks to learn functions of the form $p(d(i, j)|\{c(i, j)\})$, where $d(i, j)$ is the 3d distance between residues i and j , and $c(i, j)$ is the contact map.

4.3.2.4 Hopfield networks

A **Hopfield network** [Hop82] is a fully connected Ising model (Section 4.3.2.1) with a symmetric weight matrix, $\mathbf{W} = \mathbf{W}^\top$. The corresponding energy function has the form

$$\mathcal{E}(\mathbf{x}) = -\frac{1}{2} \mathbf{x}^\top \mathbf{W} \mathbf{x} \quad (4.87)$$

where $x_i \in \{-1, +1\}$.

The main application of Hopfield networks is as an **associative memory** or **content addressable memory**. The idea is this: suppose we train on a set of fully observed bit vectors, corresponding to patterns we want to memorize. (We discuss how to do this below). Then, at test time, we present a partial pattern to the network. We would like to estimate the missing variables; this is called **pattern completion**. That is, we want to compute

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \mathcal{E}(\mathbf{x}) \quad (4.88)$$

5. To encourage the model to learn sparse connectivity, we can also compute a MAP estimate with a sparsity promoting prior, as discussed in [IM17].

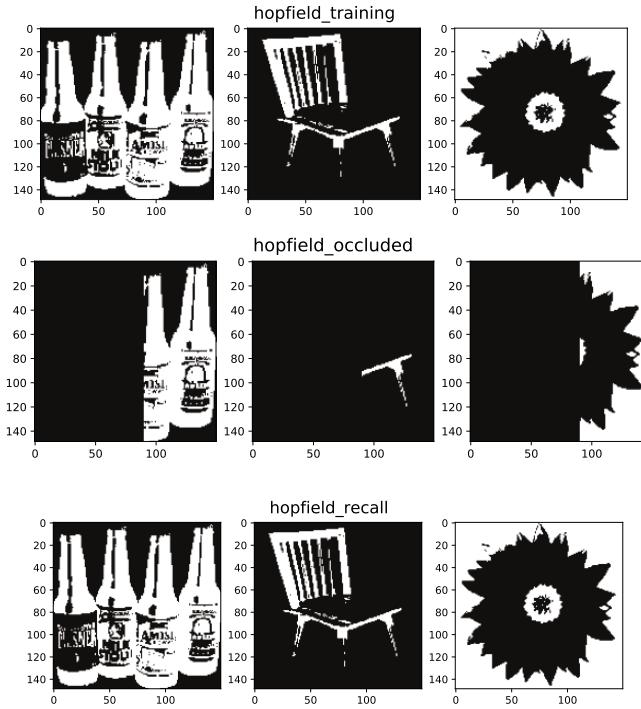


Figure 4.19: Examples of how an associative memory can reconstruct images. These are binary images of size 150×150 pixels. Top: training images. Middle row: partially visible test images. Bottom row: final state estimate. Adapted from Figure 2.1 of [HKP91]. Generated by [hopfield_demo.ipynb](#).

We can solve this optimization problem using **iterative conditional modes (ICM)**, in which we set each hidden variable to its most likely state given its neighbors. Picking the most probable state amounts to using the rule

$$\mathbf{x}^{t+1} = \text{sgn}(\mathbf{W}\mathbf{x}^t) \quad (4.89)$$

This can be seen as a deterministic version of Gibbs sampling (see Section 12.3.3).

We illustrate this process in Figure 4.19. In the top row, we show some training examples. In the middle row, we show a corrupted input, corresponding to the initial state \mathbf{x}^0 . In the bottom row, we show the final state after 30 iterations of ICM. The overall process can be thought of as retrieving a complete example from memory based on a piece of the example.

To learn the weights \mathbf{W} , we could use the maximum likelihood estimate method described in Section 4.3.9.1. (See also [HSDK12].) However, a simpler heuristic method, proposed in [Hop82], is to use the following **outer product method**:

$$\mathbf{W} = \left(\frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top \right) - \mathbf{I} \quad (4.90)$$

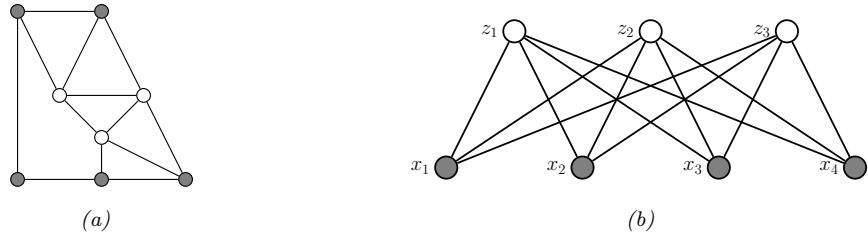


Figure 4.20: (a) A general Boltzmann machine, with an arbitrary graph structure. The shaded (visible) nodes are partitioned into input and output, although the model is actually symmetric and defines a joint distribution on all the nodes. (b) A restricted Boltzmann machine with a bipartite structure. Note the lack of intra-layer connections.

This normalizes the output product matrix by N , and then sets the diagonal to 0. This ensures the energy is low for patterns that match any of the examples in the training set. This is the technique we used in Figure 4.19. Note, however, that this method not only stores the original patterns but also their inverses, and other linear combinations. Consequently there is a limit to how many examples the model can store before they start to ‘‘collide’’ in the memory. Hopfield proved that, for random patterns, the network capacity is $\sim 0.14N$.

4.3.3 MRFs with latent variables (Boltzmann machines, etc.)

In this section, we discuss MRFs which contain latent variables, as a way to represent high dimensional joint distributions in discrete spaces.

4.3.3.1 Vanilla Boltzmann machines

MRFs in which all the variables are visible are limited in their expressive power, since the only way to model correlation between the variables is by directly adding an edge. An alternative approach is to introduce latent variables. A **Boltzmann machine** [AHS85] is like an Ising model (Section 4.3.2.1) with latent variables. In addition, the graph structure can be arbitrary (not just a lattice), and the binary states are $x_i \in \{0, 1\}$ instead of $x_i \in \{-1, +1\}$. We usually partition the nodes into hidden nodes z and visible nodes x , as shown in Figure 4.20(a).

4.3.3.2 Restricted Boltzmann machines (RBMs)

Unfortunately, exact inference (and hence learning) in Boltzmann machines is intractable, and even approximate inference (e.g., Gibbs sampling, Section 12.3) can be slow. However, suppose we restrict the architecture so that the nodes are arranged in two layers, and so that there are no connections between nodes within the same layer (see Figure 4.20(b)). This model is known as a **restricted Boltzmann machine (RBM)** [HT01; HS06a], or a **harmonium** [Smo86]. The RBM supports efficient approximate inference, since the hidden nodes are conditionally independent given the visible nodes, i.e., $p(z|x) = \prod_{k=1}^K p(z_k|x)$. Note this is in contrast to a directed two-layer models, where the explaining away effect causes the latent variables to become ‘‘entangled’’ in the posterior even if they are independent in the prior.

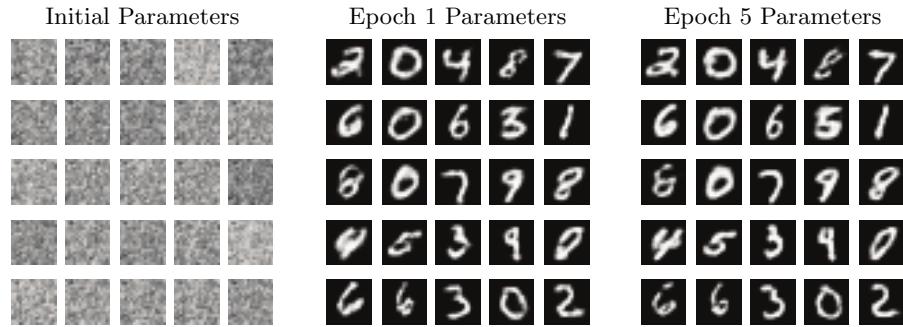


Figure 4.21: Some reconstructed images generated by a binary RBM fit to MNIST. Generated by `rbm_contrastive_divergence.ipynb`.

Visible	Hidden	Name	Reference
Binary	Binary	Binary RBM	[HS06a]
Gaussian	Binary	Gaussian RBM	[WS05]
Categorical	Binary	Categorical RBM	[SMH07]
Multiple categorical	Binary	Replicated softmax/undirected LDA	[SH10]
Gaussian	Gaussian	Undirected PCA	[MM01]
Binary	Gaussian	Undirected binary PCA	[WS05]

Table 4.4: Summary of different kinds of RBM.

Typically the hidden and visible nodes in an RBM are binary, so the energy terms have the form $w_{dk}x_dz_k$. If $z_k = 1$, then the k 'th hidden unit adds a term of the form $\mathbf{w}_k^\top \mathbf{x}$ to the energy; this can be thought of as a “soft constraint”. If $z_k = 0$, the hidden unit is not active, and does not have an opinion about this data example. By turning on different combinations of constraints, we can create complex distributions on the visible data. This is an example of a **product of experts** (Section 24.1.1), since $p(\mathbf{x}|\mathbf{z}) = \prod_{k:z_k=1} \exp(\mathbf{w}_k^\top \mathbf{x})$.

This can be thought of as a mixture model with an exponential number of hidden components, corresponding to 2^H settings of \mathbf{z} . That is, \mathbf{z} is a **distributed representation**, whereas a standard mixture model uses a **localist representation**, where $z \in \{1, K\}$, and each setting of z corresponds to a complete prototype or exemplar \mathbf{w}_k to which \mathbf{x} is compared, giving rise to a model of the form $p(\mathbf{x}|z = k) \propto \exp(\mathbf{w}_k^\top \mathbf{x})$.

Many different kinds of RBMs have been defined, which use different pairwise potential functions. See Table 4.4 for a summary. (Figure 4.21 gives an example of some images generated from an RBM fit to the binarized MNIST dataset.) All of these are special cases of the **exponential family harmonium** [WRZH04]. See Supplementary Section 4.3 for more details.

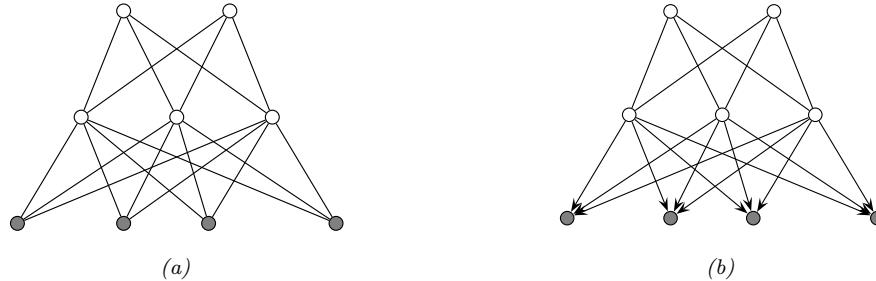


Figure 4.22: (a) Deep Boltzmann machine. (b) Deep belief network. The top two layers define the prior in terms on an RBM. The remaining layers are a directed graphical model that “decodes” the prior into observable data.

4.3.3.3 Deep Boltzmann machines

We can make a “deep” version of an RBM by stacking multiple layers; this is called a **deep Boltzmann machine** [SH09]. For example, the two layer model in Figure 4.22(a) has the form

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2 | \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{W}_1, \mathbf{W}_2)} \exp (\mathbf{x}^\top \mathbf{W}_1 \mathbf{z}_1 + \mathbf{z}_1^\top \mathbf{W}_2 \mathbf{z}_2) \quad (4.91)$$

where \mathbf{x} are the visible nodes at the bottom, and we have dropped bias terms for brevity.

4.3.3.4 Deep belief networks (DBNs)

We can use an RBM as a prior over a latent distributed code, and then use a DPGM “decoder” to convert this into the observed data, as shown in Figure 4.22(b). The corresponding joint distribution has the form

$$p(\mathbf{x}, \mathbf{z}_1, \mathbf{z}_2 | \boldsymbol{\theta}) = p(\mathbf{x} | \mathbf{z}_1, \mathbf{W}_1) \frac{1}{Z(\mathbf{W}_2)} \exp (\mathbf{z}_1^\top \mathbf{W}_2 \mathbf{z}_2) \quad (4.92)$$

In other words, it is an RBM on top of a DPGM. This combination has been called a **deep belief network (DBN)** [HOT06a]. However, this name is confusing, since it is not actually a belief net. We will therefore call it a **deep Boltzmann network** (which conveniently has the same DBN abbreviation).

DBNs can be trained in a simple greedy fashion, and support fast bottom-up inference (see [HOT06a] for details). DBNs played an important role in the history of deep learning, since they were one of the first deep models that could be successfully trained. However, they are no longer widely used, since the advent of better ways to train fully supervised deep neural networks (such as using ReLU units and the Adam optimizer), and the advent of efficient ways to train deep DPGMs, such as the VAE (Section 21.2).

4.3.4 Maximum entropy models

In Section 2.4.7, we show that the exponential family is the distribution with maximum entropy, subject to the constraints that the expected value of the features (sufficient statistics) $\phi(\mathbf{x})$ match

the empirical expectations. Thus the model has the form

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(\boldsymbol{\theta}^\top \phi(\mathbf{x})) \quad (4.93)$$

If the features $\phi(\mathbf{x})$ decompose according to a graph structure, we get a kind of MRF known as a **maximum entropy model**. We give some examples below.

4.3.4.1 Log-linear models

Suppose the potential functions have the following log-linear form:

$$\psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c) = \exp(\boldsymbol{\theta}_c^\top \phi(\mathbf{x}_c)) \quad (4.94)$$

where $\phi(\mathbf{x}_c)$ is a feature vector derived from the variables in clique c . Then the overall model is given by

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_c \boldsymbol{\theta}_c^\top \phi(\mathbf{x}_c)\right) \quad (4.95)$$

For example, in a Gaussian graphical model (GGM), we have

$$\phi([x_i, x_j]) = [x_i, x_j, x_i x_j] \quad (4.96)$$

for $x_i \in \mathbb{R}$. And in an Ising model, we have

$$\phi([x_i, x_j]) = [x_i, x_j, x_i x_j] \quad (4.97)$$

for $x_i \in \{-1, +1\}$. Thus both of these are maxent models. However, there are two key differences: first, in a GGM, the variables are real-valued, not binary; second, in a GGM, the partition function $Z(\boldsymbol{\theta})$ can be computed in $O(D^3)$ time, whereas in a Boltzmann machine, computing the partition function can take $O(2^D)$ time (see Section 9.5.4 for details).

If the features ϕ are structured in a hierarchical way (capturing first order interactions, and second order interactions, etc.), and all the variables \mathbf{x} are categorical, the resulting model is known in statistics as a **log-linear model**. However, in the ML community, the term “log-linear model” is often used to describe any model of the form Equation (4.95).

4.3.4.2 Feature induction for a maxent spelling model

In some applications, we assume the features $\phi(\mathbf{x})$ are known. However, it is possible to learn the features in a maxent model in an unsupervised way; this is known as **feature induction**.

A common approach to feature induction, first proposed in [DDL97; ZWM97], is to start with a base set of features, and then to continually create new feature combinations out of old ones, greedily adding the best ones to the model.

As an example of this approach, [DDL97] describe how to build models to represent English spelling. This can be formalized as a probability distribution over variable length strings, $p(\mathbf{x}|\boldsymbol{\theta})$,

where x_t is a letter in the English alphabet. Initially the model has no features, which represents the uniform distribution. The algorithm starts by choosing to add the feature

$$\phi_1(\mathbf{x}) = \sum_i \mathbb{I}(x_i \in \{a, \dots, z\}) \quad (4.98)$$

which checks if any letter is lowercase or not. After the feature is added, the parameters are (re)-fit by maximum likelihood (a computationally difficult problem, which we discuss in Section 4.3.9.1). For this feature, it turns out that $\hat{\theta}_1 = 1.944$, which means that a word with a lowercase letter in any position is about $e^{1.944} \approx 7$ times more likely than the same word without a lowercase letter in that position. Some samples from this model, generated using (annealed) Gibbs sampling (described in Section 12.3), are shown below.⁶

```
m, r, xevo, iijiir, b, to, jz, gsr, wq, vf, x, ga, msmGh, pcp, d, oziVlal, hzagh, yzop, io,
advzmxnv, ijk_bolft, x, emx, kayerf, mlj, rawzyb, jp, ag, ctdnnnbg, wgdw, t, kguv, cy,
spxcq, uzflbbf, dxtkkn, cxwx, jpd, ztzh, lv, zhpkvnu, l^, r, qee, nynrx, atze4n, ik, se, w,
lrh, hp+, yrqyka'h, zcnegotcnx, igcump, zjcjs, lqpWiQu, cefmfhc, o, lb, fdcY, tzby, yopxmvk,
by, fz, t, govyccm, ijyiduwfzo, 6xr, duh, ejv, pk, pjw, l, fl, w
```

The second feature added by the algorithm checks if two adjacent characters are lowercase:

$$\phi_2(\mathbf{x}) = \sum_{i \sim j} \mathbb{I}(x_i \in \{a, \dots, z\}, x_j \in \{a, \dots, z\}) \quad (4.99)$$

Now the model has the form

$$p(\mathbf{x}) = \frac{1}{Z} \exp(\theta_1 \phi_1(\mathbf{x}) + \theta_2 \phi_2(\mathbf{x})) \quad (4.100)$$

Continuing in this way, the algorithm adds features for the strings `s>` and `ing>`, where `>` represents the end of word, and for various regular expressions such as `[0-9]`, etc. Some samples from the model with 1000 features, generated using (annealed) Gibbs sampling, are shown below.

```
was, reaser, in, there, to, will, , was, by, homes, thing, be, reloverated, ther, which,
conists, at, fores, anditing, with, Mr., proveral, the, , ***, on't, prolling, prothere, ,
mento, at, yaou, 1, chestraing, for, have, to, intrally, of, qut, ., best, compers, ***, 
cluseliment, uster, of, is, deveral, this, thise, of, offect, inatever, thifer,
constrained, stater, vill, in, thase, in, youse, menttering, and, ., of, in, verate, of,
to
```

If we define a feature for every possible combination of letters, we can represent any probability distribution. However, this will overfit. The power of the maxent approach is that we can choose which features matter for the domain.

An alternative approach is to introduce latent variables, that implicitly model correlations amongst the visible nodes, rather than explicitly having to learn feature functions. See Section 4.3.3 for an example of such a model.

6. We thank John Lafferty for sharing this example.

4.3.5 Gaussian MRFs

In Section 4.2.3, we showed how to represent a multivariate Gaussian using a DPGM. In this section, we show how to represent a multivariate Gaussian using a UPGM. (For further details on GMRFs, see e.g., [RH05].)

4.3.5.1 Standard GMRFs

A **Gaussian graphical model** (or **GGM**), also called a **Gaussian MRF**, is a pairwise MRF of the following form:

$$p(\mathbf{x}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{i \sim j} \psi_{ij}(x_i, x_j) \prod_i \psi_i(x_i) \quad (4.101)$$

$$\psi_{ij}(x_i, x_j) = \exp\left(-\frac{1}{2}x_i \Lambda_{ij} x_j\right) \quad (4.102)$$

$$\psi_i(x_i) = \exp\left(-\frac{1}{2}\Lambda_{ii}x_i^2 + \eta_i x_i\right) \quad (4.103)$$

$$Z(\boldsymbol{\theta}) = (2\pi)^{D/2} |\Lambda|^{-\frac{1}{2}} \quad (4.104)$$

The ψ_{ij} are **edge potentials** (pairwise terms), and each the ψ_i are **node potentials** or **unary terms**. (We could absorb the unary terms into the pairwise terms, but we have kept them separate for clarity.)

The joint distribution can be rewritten in a more familiar form as follows:

$$p(\mathbf{x}) \propto \exp[\boldsymbol{\eta}^\top \mathbf{x} - \frac{1}{2} \mathbf{x}^\top \boldsymbol{\Lambda} \mathbf{x}] \quad (4.105)$$

This is called the **information form** of a Gaussian; $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{\eta} = \boldsymbol{\Lambda} \boldsymbol{\mu}$ are called the **canonical parameters**.

If $\Lambda_{ij} = 0$, there is no pairwise term connecting x_i and x_j , and hence $x_i \perp x_j | \mathbf{x}_{-ij}$, where \mathbf{x}_{-ij} are all the nodes except for x_i and x_j . Hence the zero entries in $\boldsymbol{\Lambda}$ are called **structural zeros**. This means we can use ℓ_1 regularization on the weights to learn a sparse graph, a method known as **graphical lasso** (see Supplementary Section 30.4.2).

Note that the covariance matrix $\boldsymbol{\Sigma} = \boldsymbol{\Lambda}^{-1}$ can be dense even if the precision matrix $\boldsymbol{\Lambda}$ is sparse. For example, consider an AR(1) process with correlation parameter ρ .⁷ The precision matrix (for a graph with $T = 7$ nodes) looks like this:

$$\boldsymbol{\Lambda} = \frac{1}{\tau^2} \begin{pmatrix} 1 & -\rho & & & & & \\ -\rho & 1 + \rho^2 & -\rho & & & & \\ & -\rho & 1 + \rho^2 & -\rho & & & \\ & & -\rho & 1 + \rho^2 & -\rho & & \\ & & & -\rho & 1 + \rho^2 & -\rho & \\ & & & & -\rho & 1 + \rho^2 & -\rho \\ & & & & & -\rho & 1 \end{pmatrix} \quad (4.106)$$

7. This example is from <https://dansblog.netlify.app/posts/2022-03-22-a-linear-mixed-effects-model/>.

But the covariance matrix is fully dense:

$$\boldsymbol{\Lambda}^{-1} = \tau^2 \begin{pmatrix} \rho & \rho^2 & \rho^3 & \rho^4 & \rho^5 & \rho^6 & \rho^7 \\ \rho^2 & \rho & \rho^2 & \rho^3 & \rho^4 & \rho^5 & \rho^6 \\ \rho^3 & \rho^2 & \rho & \rho^2 & \rho^3 & \rho^4 & \rho^5 \\ \rho^4 & \rho^3 & \rho^2 & \rho & \rho^2 & \rho^3 & \rho^4 \\ \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & \rho^2 & \rho^3 \\ \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho & \rho^2 \\ \rho^7 & \rho^6 & \rho^5 & \rho^4 & \rho^3 & \rho^2 & \rho \end{pmatrix} \quad (4.107)$$

This follows because, in a chain structured UPGM, every pair of nodes is marginally correlated, even if they may be conditionally independent given a separator.

4.3.5.2 Nonlinear Gaussian MRFs

In this section, we consider a generalization of GGMs to handle the case of nonlinear models. Suppose the joint is given by a product of local factors, or clique potentials, ψ_c , each of which is defined on a set or clique variables \mathbf{x}_c as follows:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_c \psi_c(\mathbf{x}_c) \quad (4.108)$$

$$\psi_c(\mathbf{x}_c) = \exp(-E_c(\mathbf{x}_c)) \quad (4.109)$$

$$E_c(\mathbf{x}_c) = \frac{1}{2}(f_c(\mathbf{x}_c) - \mathbf{d}_c)^\top \boldsymbol{\Sigma}_c^{-1}(f_c(\mathbf{x}_c) - \mathbf{d}_c) \quad (4.110)$$

where \mathbf{d}_c is an optional local evidence term for the c 'th clique, and f_c is some measurement function.

Suppose the measurement function f_c is linear, i.e.,

$$f_c(\mathbf{x}) = \mathbf{J}_c \mathbf{x} + \mathbf{b}_c \quad (4.111)$$

In this case, the energy for clique c becomes

$$E_c(\mathbf{x}_c) = \frac{1}{2} \mathbf{x}_c^\top \underbrace{\mathbf{J}_c^\top \boldsymbol{\Sigma}_c^{-1} \mathbf{J}_c}_{\boldsymbol{\Lambda}_c} \mathbf{x}_c + \mathbf{x}_c^\top \underbrace{\mathbf{J}_c^\top \boldsymbol{\Sigma}_c^{-1} (\mathbf{b}_c - \mathbf{d}_c)}_{-\boldsymbol{\eta}_c} + \underbrace{\frac{1}{2} (\mathbf{b}_c - \mathbf{d}_c)^\top \boldsymbol{\Sigma}_c^{-1} (\mathbf{b}_c - \mathbf{d}_c)}_{k_c} \quad (4.112)$$

$$= \frac{1}{2} \mathbf{x}_c^\top \boldsymbol{\Lambda}_c \mathbf{x}_c - \boldsymbol{\eta}_c^\top \mathbf{x}_c + k_c \quad (4.113)$$

which is a standard Gaussian factor. If f_c is nonlinear, it is common to linearize the model around the current estimate \mathbf{x}_c^0 to get

$$f_c(\mathbf{x}_c) \approx f_c(\mathbf{x}_c^0) + \mathbf{J}_c(\mathbf{x}_c - \mathbf{x}_c^0) = \mathbf{J}_c \mathbf{x}_c + \underbrace{(f_c(\mathbf{x}_c^0) - \mathbf{J}_c \mathbf{x}_c^0)}_{\mathbf{b}_c} \quad (4.114)$$

where \mathbf{J}_c is the Jacobian of $f_c(\mathbf{x}_c)$ wrt \mathbf{x}_c . This gives us a “temporary” Gaussian factor that we can use for inference. This process can be iterated for improved accuracy.

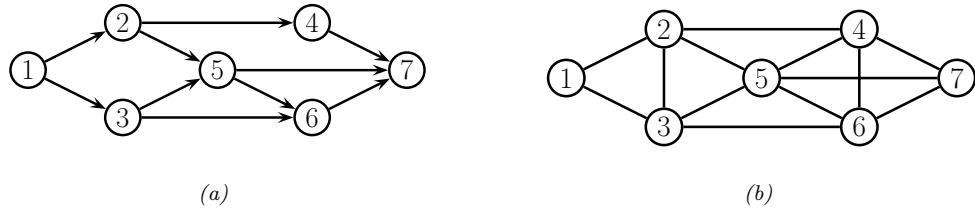


Figure 4.23: (a) A DPGM. (b) Its moralized version, represented as a UPGM.

4.3.6 Conditional independence properties

In this section, we explain how UPGMs encode conditional independence assumptions.

4.3.6.1 Basic results

UPGMs define CI relationships via simple graph separation as follows: given 3 sets of nodes A , B , and C , we say $\mathbf{X}_A \perp_G \mathbf{X}_B | \mathbf{X}_C$ iff C separates A from B in the graph G . This means that, when we remove all the nodes in C , if there are no paths connecting any node in A to any node in B , then the CI property holds. This is called the **global Markov property** for UPGMs. For example, in Figure 4.23(b), we have that $\{X_1, X_2\} \perp \{X_6, X_7\} | \{X_3, X_4, X_5\}$.

The smallest set of nodes that renders a node t conditionally independent of all the other nodes in the graph is called t 's **Markov blanket**; we will denote this by $\text{mb}(t)$. Formally, the Markov blanket satisfies the following property:

$$t \perp \mathcal{V} \setminus \text{cl}(t) | \text{mb}(t) \quad (4.115)$$

where $\text{cl}(t) \triangleq \text{mb}(t) \cup \{t\}$ is the **closure** of node t , and $\mathcal{V} = \{1, \dots, N_G\}$ is the set of all nodes. One can show that, in a UPGM, a node's Markov blanket is its set of immediate neighbors. This is called the **undirected local Markov property**. For example, in Figure 4.23(b), we have $\text{mb}(X_5) = \{X_2, X_3, X_4, X_6, X_7\}$.

From the local Markov property, we can easily see that two nodes are conditionally independent given the rest if there is no direct edge between them. This is called the **pairwise Markov property**. In symbols, this is written as

$$s \perp t | \mathcal{V} \setminus \{s, t\} \iff G_{st} = 0 \quad (4.116)$$

where $G_{st} = 0$ means there is no edge between s and t (so there is a 0 in the adjacency matrix).

Using the three Markov properties we have discussed, we can derive the following CI properties (amongst others) from the UPGM in Figure 4.23(b): $X_1 \perp X_7|\text{rest}$ (pairwise); $X_1 \perp \text{rest}|X_2, X_3$ (local); $X_1, X_2 \perp X_6, X_7|X_3, X_4, X_5$ (global).

It is obvious that global Markov implies local Markov which implies pairwise Markov. What is less obvious is that pairwise implies global, and hence that all these Markov properties are the same, as illustrated in Figure 4.24 (see e.g., [KF09a, p119] for a proof).⁸ The importance of this result is that it is usually easier to empirically assess pairwise conditional independence; such pairwise CI statements can be used to construct a graph from which global CI statements can be extracted.

⁸. This assumes $p(\mathbf{x}) > 0$ for all \mathbf{x} , i.e., that p is a positive density. The restriction to positive densities arises because

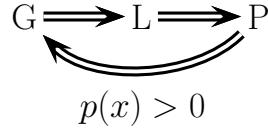


Figure 4.24: Relationship between Markov properties of UPGMs.

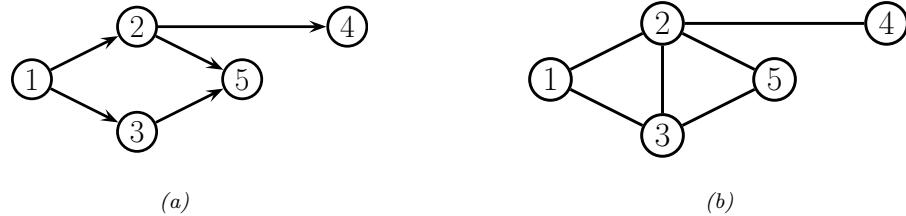


Figure 4.25: (a) The ancestral graph induced by the DAG in Figure 4.23(a) wrt $U = \{X_2, X_4, X_5\}$. (b) The moralized version of (a).

4.3.6.2 An undirected alternative to d-separation

We have seen that determining CI relationships in UPGMs is much easier than in DPGMs, because we do not have to worry about the directionality of the edges. That is, we can use simple graph separation, instead of d-separation.

In this section, we show how to convert a DPGM to a UPGM, so that we can infer CI relationships for the DPGM using simple graph separation. It is tempting to simply convert the DPGM to a UPGM by dropping the orientation of the edges, but this is clearly incorrect, since a v-structure $A \rightarrow B \leftarrow C$ has quite different CI properties than the corresponding undirected chain $A - B - C$ (e.g., the latter graph incorrectly states that $A \perp C|B$). To avoid such incorrect CI statements, we can add edges between the “unmarried” parents A and C , and then drop the arrows from the edges, forming (in this case) a fully connected undirected graph. This process is called **moralization**. Figure 4.23 gives a larger example of moralization: we interconnect 2 and 3, since they have a common child 5, and we interconnect 4, 5, and 6, since they have a common child 7.

Unfortunately, moralization loses some CI information, and therefore we cannot use the moralized UPGM to determine CI properties of the DPGM. For example, in Figure 4.23(a), using d-separation, we see that $X_4 \perp X_5|X_2$. Adding a moralization arc $X_4 - X_5$ would lose this fact (see Figure 4.23(b)). However, notice that the 4-5 moralization edge, due to the common child 7, is not needed if we do not observe 7 or any of its descendants. This suggests the following approach to determining if $A \perp B|C$. First we form the **ancestral graph** of DAG G with respect to $U = A \cup B \cup C$. This means we remove all nodes from G that are not in U or are not ancestors of U . We then moralize this ancestral

deterministic constraints can result in independencies present in the distribution that are not explicitly represented in the graph. See e.g., [KF09a, p120] for some examples. Distributions with non-graphical CI properties are said to be **unfaithful** to the graph, so $I(p) \neq I(G)$.

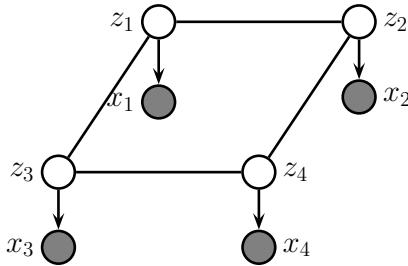


Figure 4.26: A grid-structured MRF with hidden nodes z_i and local evidence nodes x_i . The prior $p(\mathbf{z})$ is an undirected Ising model, and the likelihood $p(\mathbf{x}|\mathbf{z}) = \prod_i p(x_i|z_i)$ is a directed fully factored model.

graph, and apply the simple graph separation rules for UPGMs. For example, in Figure 4.25(a), we show the ancestral graph for Figure 4.23(a) using $U = \{X_2, X_4, X_5\}$. In Figure 4.25(b), we show the moralized version of this graph. It is clear that we now correctly conclude that $X_4 \perp X_5 | X_2$.

4.3.7 Generation (sampling)

Unlike with DPGMs, it can be quite slow to sample from an UPGM, even from the unconditional prior, because there is no ordering of the variables. Furthermore, we cannot easily compute the probability of any configuration unless we know the value of Z . Consequently it is common to use MCMC methods for generating from an UPGM (see Chapter 12).

In the special case of UPGMs with low treewidth and discrete or Gaussian potentials, it is possible to use the junction tree algorithm to draw samples using dynamic programming (see Supplementary Section 9.2.3).

4.3.8 Inference

We discuss inference in graphical models in detail in Chapter 9. In this section, we just give an example.

Suppose we have an image composed of binary pixels, z_i , but we only observe noisy versions of the pixels, x_i . We assume the joint model has the form

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \left[\frac{1}{Z} \sum_{i \sim j} \psi_{ij}(z_i, z_j) \right] \prod_i p(x_i|z_i) \quad (4.117)$$

where $p(\mathbf{z})$ is an Ising model prior, and $p(x_i|z_i) = \mathcal{N}(x_i|z_i, \sigma^2)$, for $z_i \in \{-1, +1\}$. This model uses a UPGM as a prior, and has directed edges for the likelihood, as shown in Figure 4.26; such a hybrid undirected-directed model is called a **chain graph** (even though it is not chain-structured).

The inference task is to compute the posterior marginals $p(z_i|\mathbf{x})$, or the posterior MAP estimate, $\text{argmax}_{\mathbf{z}} p(\mathbf{z}|\mathbf{x})$. The exact computation is intractable for large grids (for reasons explained in Section 9.5.4), so we must use approximate methods. There are many algorithms that we can use, including mean field variational inference (Section 10.3.2), Gibbs sampling (Section 12.3.3), loopy belief propagation (Section 9.4), etc. In Figure 4.27, we show the results of variational inference.



Figure 4.27: Example of image denoising using mean field variational inference. We use an Ising prior with $W_{ij} = 1$ and a Gaussian noise model with $\sigma = 2$. (a) Noisy image. (b) Result of inference. Generated by `ising_image_denoise_demo.ipynb`.

4.3.9 Learning

In this section, we discuss how to estimate the parameters for an MRF. As we will see, computing the MLE can be computationally expensive, even in the fully observed case, because of the need to deal with the partition function $Z(\boldsymbol{\theta})$. And computing the posterior over the parameters, $p(\boldsymbol{\theta}|\mathcal{D})$, is even harder, because of the additional normalizing constant $p(\mathcal{D})$ — this case has been called **doubly intractable** [MGM06]. Consequently we will focus on point estimation methods such as MLE and MAP. (For one approach to Bayesian parameter inference in an MRF, based on persistent variational inference, see [IM17].)

4.3.9.1 Learning from complete data

We will start by assuming there are no hidden variables or missing data during training (this is known as the **complete data** setting). For simplicity of presentation, we restrict our discussion to the case of MRFs with log-linear potential functions. (See Section 24.2 for the general nonlinear case, where we discuss MLE for energy-based models.)

In particular, we assume the distribution has the following form:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_c \boldsymbol{\theta}_c^\top \phi_c(\mathbf{x})\right) \quad (4.118)$$

where c indexes the cliques. The (averaged) log-likelihood of the full dataset becomes

$$\ell(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_n \log p(\mathbf{x}_n|\boldsymbol{\theta}) = \frac{1}{N} \sum_n \left[\sum_c \boldsymbol{\theta}_c^\top \phi_c(\mathbf{x}_n) - \log Z(\boldsymbol{\theta}) \right] \quad (4.119)$$

Its gradient is given by

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N} \sum_n \left[\phi_c(\mathbf{x}_n) - \frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z(\boldsymbol{\theta}) \right] \quad (4.120)$$

We know from Section 2.4.3 that the derivative of the log partition function wrt $\boldsymbol{\theta}_c$ is the expectation

of the c 'th feature vector under the model, i.e.,

$$\frac{\partial \log Z(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_c} = \mathbb{E} [\phi_c(\mathbf{x}) | \boldsymbol{\theta}] = \sum_{\mathbf{x}} p(\mathbf{x} | \boldsymbol{\theta}) \phi_c(\mathbf{x}) \quad (4.121)$$

Hence the gradient of the log likelihood is

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N} \sum_n [\phi_c(\mathbf{x}_n)] - \mathbb{E} [\phi_c(\mathbf{x})] \quad (4.122)$$

When the expected value of the features according to the data is equal to the expected value of the features according to the model, the gradient will be zero, so we get

$$\mathbb{E}_{p_D} [\phi_c(\mathbf{x})] = \mathbb{E}_{p(\mathbf{x} | \boldsymbol{\theta})} [\phi_c(\mathbf{x})] \quad (4.123)$$

This is called **moment matching**. Evaluating the $\mathbb{E}_{p_D} [\phi_c(\mathbf{x})]$ term is called the **clamped phase** or **positive phase**, since \mathbf{x} is set to the observed values \mathbf{x}_n ; evaluating the $\mathbb{E}_{p(\mathbf{x} | \boldsymbol{\theta})} [\phi_c(\mathbf{x})]$ term is called the **unclamped phase** or **negative phase**, since \mathbf{x} is free to vary, and is generated by the model.

In the case of MRFs with tabular potentials (i.e., one feature per entry in the clique table), we can use an algorithm called **iterative proportional fitting** or **IPF** [Fie70; BFH75; JP95] to solve these equations in an iterative fashion.⁹ But in general, we must use gradient methods to perform parameter estimation.

4.3.9.2 Computational issues

The biggest computational bottleneck in fitting MRFs and CRFs using MLE is the cost of computing the derivative of the log partition function, $\log Z(\boldsymbol{\theta})$, which is needed to compute the derivative of the log likelihood, as we saw in Section 4.3.9.1. To see why this is slow to compute, note that

$$\nabla_{\boldsymbol{\theta}} \log Z(\boldsymbol{\theta}) = \frac{\nabla_{\boldsymbol{\theta}} Z(\boldsymbol{\theta})}{Z(\boldsymbol{\theta})} = \frac{1}{Z(\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \int \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) d\mathbf{x} = \frac{1}{Z(\boldsymbol{\theta})} \int \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) d\mathbf{x} \quad (4.124)$$

$$= \frac{1}{Z(\boldsymbol{\theta})} \int \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) d\mathbf{x} = \int \frac{\tilde{p}(\mathbf{x}; \boldsymbol{\theta})}{Z(\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) d\mathbf{x} \quad (4.125)$$

$$= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x}; \boldsymbol{\theta})} [\nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta})] \quad (4.126)$$

where in Equation (4.125) we used the fact that $\nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{\tilde{p}(\mathbf{x}; \boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x}; \boldsymbol{\theta})$ (this is known as the **log-derivative trick**). Thus we see that we need to draw samples from the model at each step of SGD training, just to estimate the gradient.

In Section 24.2.1, we discuss various efficient sampling methods. However, it is also possible to use alternative estimators which do not use the principle of maximum likelihood. For example, in Section 24.2.2 we discuss the technique of contrastive divergence. And in Section 4.3.9.3, we discuss the technique of pseudolikelihood. (See also [Sto17] for a review of many methods for parameter estimation in MRFs.)

9. In the case of decomposable graphs, IPF converges in a single iteration. Intuitively, this is because a decomposable graph can be converted to a DAG without any loss of information, as explained in Section 4.5, and we know that we can compute the MLE for tabular CPDs in closed form, just by normalizing the counts.



Figure 4.28: (a) A small 2d lattice. (b) The representation used by pseudo likelihood. Solid nodes are observed neighbors. Adapted from Figure 2.2 of [Car03].

4.3.9.3 Maximum pseudolikelihood estimation

When fitting fully visible MRFs (or CRFs), a simple alternative to maximizing the likelihood is to maximize the **pseudo likelihood** [Bes75], defined as follows:

$$\ell_{PL}(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_{n=1}^N \sum_{d=1}^D \log p(x_{nd} | \mathbf{x}_{n,-d}, \boldsymbol{\theta}) \quad (4.127)$$

That is, we optimize the product of the full conditionals, also known as the **composite likelihood** [Lin88a; DL10; VRF11]. Compare this to the objective for maximum likelihood:

$$\ell_{ML}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \log p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (4.128)$$

In the case of Gaussian MRFs, PL is equivalent to ML [Bes75], although this is not true in general. Nevertheless, it is a consistent estimator in the large sample limit [LJ08].

The PL approach is illustrated in Figure 4.28 for a 2d grid. We learn to predict each node, given all of its neighbors. This objective is generally fast to compute since each full conditional $p(x_d | \mathbf{x}_{-d}, \boldsymbol{\theta})$ only requires summing over the states of a single node, x_d , in order to compute the local normalization constant. The PL approach is similar to fitting each full conditional separately, except that, in PL, the parameters are tied between adjacent nodes.

Experiments in [PW05; HT09] suggest that PL works as well as exact ML for fully observed Ising models, but is much faster. In [Eke+13], they use PL to fit Potts models to (aligned) protein sequence data. However, when fitting RBMs, [Mar+10] found that PL is worse than some of the stochastic ML methods we discuss in Section 24.2.

Another more subtle problem is that each node assumes that its neighbors have known values during training. If node $j \in \text{nbr}(i)$ is a perfect predictor for node i (where $\text{nbr}(i)$ is the set of neighbors), then j will learn to rely completely on node i , even at the expense of ignoring other potentially useful information, such as its local evidence, say y_i . At test time, the neighboring nodes will not be observed, and performance will suffer.¹⁰

10. Geoff Hinton has an analogy for this problem. Suppose we want to learn to denoise images of symmetric shapes, such as Greek vases. Each hidden pixel x_i depends on its spatial neighbors, as well as the noisy observation y_i . Since its symmetric counterpart x_j will perfectly predict x_i , the model will ignore y_i and just rely on x_j , even though x_j will not be available at test time.

4.3.9.4 Learning from incomplete data

In this section, we consider parameter estimation for MRFs (and CRFs) with hidden variables. Such **incomplete data** can arise for several reasons. For example, we may want to learn a model of the form $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ which lets us infer a “clean” image \mathbf{z} from a noisy or corrupted version \mathbf{x} . If we only observe \mathbf{x} , the model is called a **hidden Gibbs random field**. See Section 10.3.2 for an example. As another example, we may have a CRF in which the hidden variables are used to encode an unknown alignment between the inputs and outputs [Qua+07], or to model missing parts of the input [SRS10].

We now discuss how to compute the MLE in such cases. For notational simplicity, we focus on unconditional models (MRFs, not CRFs), and we assume all the potentials are log-linear. In this case, the model has the following form:

$$p(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{z}))}{Z(\boldsymbol{\theta})} = \frac{\tilde{p}(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})}{Z(\boldsymbol{\theta})} \quad (4.129)$$

$$Z(\boldsymbol{\theta}) = \sum_{\mathbf{x}, \mathbf{z}} \exp(\boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}, \mathbf{z})) \quad (4.130)$$

where $\tilde{p}(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})$ is the unnormalized distribution. We have dropped the sum over cliques c for brevity.

The log likelihood is now given by

$$\ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \log \left(\sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\theta}) \right) \quad (4.131)$$

$$= \frac{1}{N} \sum_{n=1}^N \log \left(\frac{1}{Z(\boldsymbol{\theta})} \sum_{\mathbf{z}_n} \tilde{p}(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\theta}) \right) \quad (4.132)$$

$$= \frac{1}{N} \sum_{n=1}^N \left[\log \sum_{\mathbf{z}_n} \tilde{p}(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\theta}) \right] - \log Z(\boldsymbol{\theta}) \quad (4.133)$$

Note that

$$\log \sum_{\mathbf{z}_n} \tilde{p}(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\theta}) = \log \sum_{\mathbf{z}_n} \exp(\boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}_n, \mathbf{z}_n)) \triangleq \log Z(\boldsymbol{\theta}, \mathbf{x}_n) \quad (4.134)$$

where $Z(\boldsymbol{\theta}, \mathbf{x}_n)$ is the same as the partition function for the whole model, except that \mathbf{x} is fixed at \mathbf{x}_n . Thus the log likelihood is a difference of two partition functions, one where \mathbf{x} is clamped to \mathbf{x}_n and \mathbf{z} is unclamped, and one where both \mathbf{x} and \mathbf{z} are unclamped. The gradient of these log partition functions corresponds to the expected features, where (in the clamped case) we condition on $\mathbf{x} = \mathbf{x}_n$. Hence

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \frac{1}{N} \sum_n [\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z} | \mathbf{x}_n, \boldsymbol{\theta})} [\boldsymbol{\phi}(\mathbf{x}_n, \mathbf{z})]] - \mathbb{E}_{(\mathbf{z}, \mathbf{x}) \sim p(\mathbf{z}, \mathbf{x} | \boldsymbol{\theta})} [\boldsymbol{\phi}(\mathbf{x}, \mathbf{z})] \quad (4.135)$$

4.4 Conditional random fields (CRFs)

A **conditional random field** or **CRF** [LMP01] is a Markov random field defined on a set of related label nodes \mathbf{y} , whose joint probability is predicted conditional on a fixed set of input nodes \mathbf{x} . More

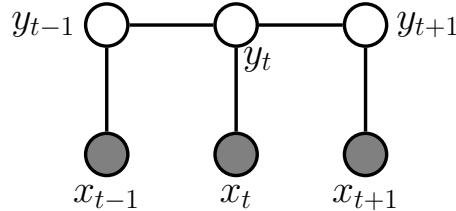


Figure 4.29: A 1d conditional random field (CRF) for sequence labeling.

precisely, it corresponds to a model of the following form:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{x}, \boldsymbol{\theta})} \prod_c \psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) \quad (4.136)$$

(Note how the partition function now depends on the inputs \mathbf{x} as well as the parameters $\boldsymbol{\theta}$.) Now suppose the potential functions are log-linear and have the form

$$\psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) = \exp(\boldsymbol{\theta}_c^\top \phi_c(\mathbf{x}, \mathbf{y}_c)) \quad (4.137)$$

This is a conditional version of the maxent models we discussed in Section 4.3.4. Of course, we can also use nonlinear potential functions, such as DNNs.

CRFs are useful because they capture dependencies amongst the output labels. They can therefore be used for **structured prediction**, where the output $\mathbf{y} \in \mathcal{Y}$ that we want to predict given the input \mathbf{x} lives in some structured space, such as a sequence of labels, or labels associated with nodes on a graph. In such problems, there are often constraints on the set of valid values of the output \mathbf{y} . For example, if we want to perform sentence parsing, the output should satisfy the rules of grammar (e.g., noun phrase must precede verb phrase). See Section 4.4.1 for details on the application of CRFs to NLP. In some cases, the “constraints” are “soft”, rather than “hard”. For example, if we want to associate a label with each pixel in an image (a task called semantic segmentation), we might want to “encourage” the label at one location to be the same as its neighbors, unless the visual input strongly suggests a change in semantic content at this location (e.g., at the edge of an object). See Section 4.4.2 for details on the applications of CRFs to computer vision tasks.

4.4.1 1d CRFs

In this section, we focus on 1d CRFs defined on chain-structured graphical models. The graphical model is shown in Figure 4.29. This defines a joint distribution over sequences, $\mathbf{y}_{1:T}$, given a set of inputs, $\mathbf{x}_{1:T}$, as follows:

$$p(\mathbf{y}_{1:T}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{x}, \boldsymbol{\theta})} \prod_{t=1}^T \psi(y_t, \mathbf{x}_t; \boldsymbol{\theta}) \prod_{t=2}^T \psi(y_{t-1}, y_t; \boldsymbol{\theta}) \quad (4.138)$$

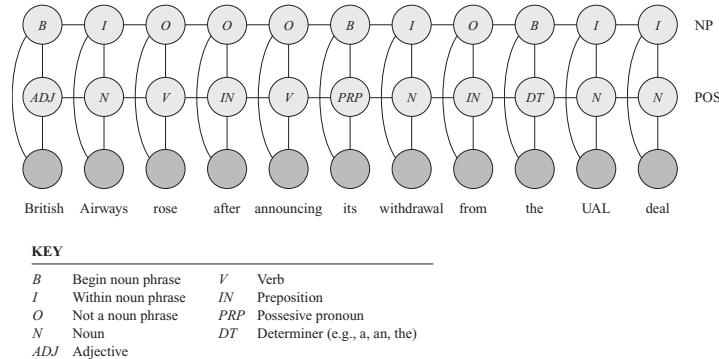


Figure 4.30: A CRF for joint part of speech tagging and noun phrase segmentation. From Figure 4.E.1 of [KF09a]. Used with kind permission of Daphne Koller.

where $\psi(y_t, \mathbf{x}_t; \boldsymbol{\theta})$ are the node potentials and $\psi(y_t, y_{t+1}; \boldsymbol{\theta})$ are the edge potentials. (We have assumed that the edge potentials are independent of the input \mathbf{x} , but this assumption is not required.)

Note that one could also consider an alternative way to define this conditional distribution, by using a discriminative directed Markov chain:

$$p(\mathbf{y}_{1:T} | \mathbf{x}, \boldsymbol{\theta}) = p(y_1 | \mathbf{x}_1; \boldsymbol{\theta}) \prod_{t=2}^T p(y_t | y_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}) \quad (4.139)$$

This is called a **maximum entropy Markov model** [MFP00]. However, it suffers from a subtle flaw compared to the CRF. In particular, in the directed model, each conditional $p(y_t | y_{t-1}, \mathbf{x}_t; \boldsymbol{\theta})$, is **locally normalized**, whereas in the CRF, the model is **globally normalized** due to the $Z(\mathbf{x}, \boldsymbol{\theta})$ term. The latter allows information to propagate through the entire sequence, as we discuss in more detail in Section 4.5.3.

CRFs were widely used in the natural language processing (NLP) community in the 1980s–2010s (see e.g., [Smi11]), although recently they have been mostly replaced by RNNs and transformers (see e.g., [Gol17]). Fortunately, we can get the best of both worlds by combining CRFs with DNNs, which allows us to combine data driven techniques with prior knowledge about constraints on the label space. We give some examples below.

4.4.1.1 Noun phrase chunking

A common task in NLP is **information extraction**, in which we try to parse a sentence into **noun phrases** (NP), such as names and addresses of people or businesses, as well as **verb phrases**, which describe who is doing what to whom (e.g., “British Airways rose”). In order to tackle this task, we can assign a **part of speech** tag to each word, where the tags correspond to Noun, Verb, Adjective, etc. In addition, to extract the span of each noun phrase, we can annotate words as being at the beginning (B) or inside (I) of a noun phrase, or outside (O) of one. See Figure 4.30 for an example.

The connections between adjacent labels can encode constraints such as the fact that B (begin) must precede I (inside). For example, the sequences OBIIO and OBIOBIO are valid (corresponding to

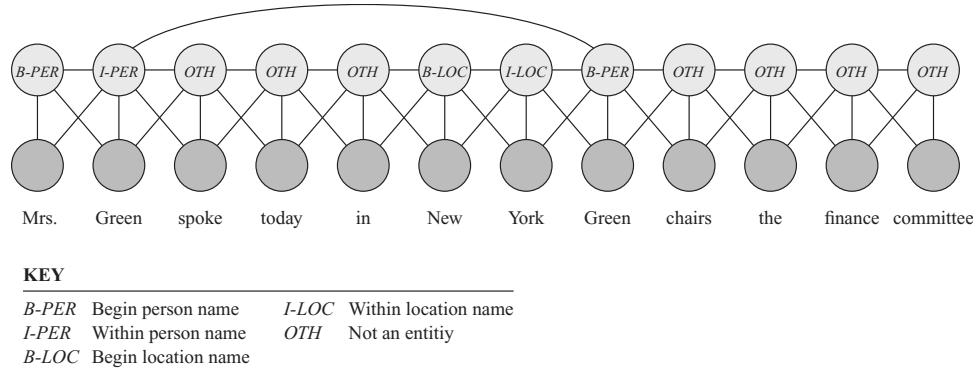


Figure 4.31: A skip-chain CRF for named entity recognition. From Figure 4.E.1 of [KF09a]. Used with kind permission of Daphne Koller.

one NP of 3 words, and two adjacent NPs of 2 words), but 0IBIO is not. This prior information can be encoded by defining $\psi(y_{t-1}^{\text{BIO}} = *, y_t^{\text{BIO}} = B, x_t; \theta)$ to be 0 for any value of * except O. We can encode similar grammatical rules for the POS tags.

Given this model, we can compute the MAP sequence of labels, and thereby extract the spans that are labeled as noun phrases. This is called **noun phrase chunking**.

4.4.1.2 Named entity recognition

In this section we consider the task of **named entity extraction**, in which we not only tag the noun phrases, but also classify them into different types. A simple approach to this is to extend the BIO notation to {B-Per, I-Per, B-Loc, I-Loc, B-Org, I-Org, Other}. However, sometimes it is ambiguous whether a word is a person, location, or something else. Proper nouns are particularly difficult to deal with because they belong to an **open class**, that is, there is an unbounded number of possible names, unlike the set of nouns and verbs, which is large but essentially fixed. For example, “British Airways” is an organization, but “British Virgin Islands” is a location.

We can get better performance by considering long-range correlations between words. For example, we might add a link between all occurrences of the same word, and force the word to have the same tag in each occurrence. (The same technique can also be helpful for resolving the identity of pronouns.) This is known as a **skip-chain CRF**. See Figure 4.31 for an illustration, where we show that the word “Green” is interpreted as a person in both occurrences within the same sentence.

We see that the graph structure itself changes depending on the input, which is an additional advantage of CRFs over generative models. Unfortunately, inference in this model is generally more expensive than in a simple chain with local connections because of the larger treewidth (see Section 9.5.2).

4.4.1.3 Natural language parsing

A generalization of chain-structured models for language is to use probabilistic grammars. In particular, a probabilistic **context free grammar** or **PCFG** in Chomsky normal form is a set of

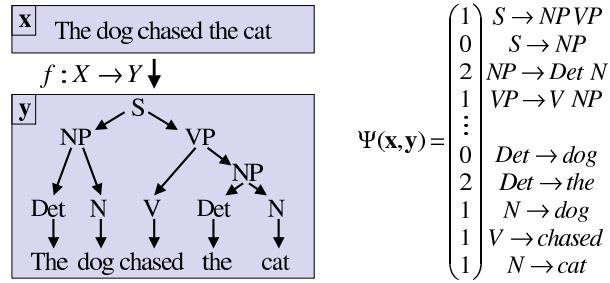


Figure 4.32: Illustration of a simple parse tree based on a context free grammar in Chomsky normal form. The feature vector $\Psi(\mathbf{x}, \mathbf{y})$ counts the number of times each production rule was used, and is used to define the energy of a particular tree structure, $E(\mathbf{y}|\mathbf{x}) = -\mathbf{w}^\top \Psi(\mathbf{x}, \mathbf{y})$. The probability distribution over trees is given by $p(\mathbf{y}|\mathbf{x}) \propto \exp(-E(\mathbf{y}|\mathbf{x}))$. From Figure 5.2 of [AHT07]. Used with kind permission of Yasemin Altun.

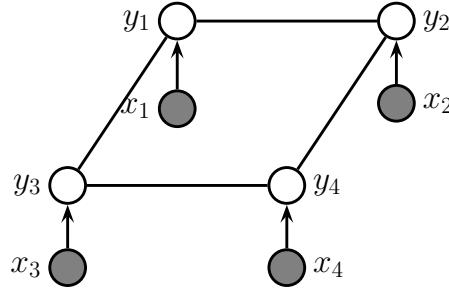


Figure 4.33: A grid-structured CRF with label nodes y_i and local evidence nodes x_i .

re-write or production rules of the form $\sigma \rightarrow \sigma'\sigma''$ or $\sigma \rightarrow x$, where $\sigma, \sigma', \sigma'' \in \Sigma$ are non-terminals (analogous to parts of speech), and $x \in \mathcal{X}$ are terminals, i.e., words. Each such rule has an associated probability. The resulting model defines a probability distribution over sequences of words. We can compute the probability of observing a particular sequence $\mathbf{x} = x_1 \dots x_T$ by summing over all trees that generate it. This can be done in $O(T^3)$ time using the **inside-outside algorithm**; see e.g., [JM08; MS99; Eis16] for details.

PCFGs are generative models. It is possible to make discriminative versions which encode the probability of a labeled tree, \mathbf{y} , given a sequence of words, \mathbf{x} , by using a CRF of the form $p(\mathbf{y}|\mathbf{x}) \propto \exp(\mathbf{w}^\top \Psi(\mathbf{x}, \mathbf{y}))$. For example, we might define $\Psi(\mathbf{x}, \mathbf{y})$ to count the number of times each production rule was used (which is analogous to the number of state transitions in a chain-structured model), as illustrated in Figure 4.32. We can also use a deep neural net to define the features, as in the **neural CRF parser** method of [DK15b].

4.4.2 2d CRFs

It is also possible to apply CRFs to image processing problems, which are usually defined on 2d grids, as illustrated in Figure 4.33. (Compare this to the generative model in Figure 4.26.) This

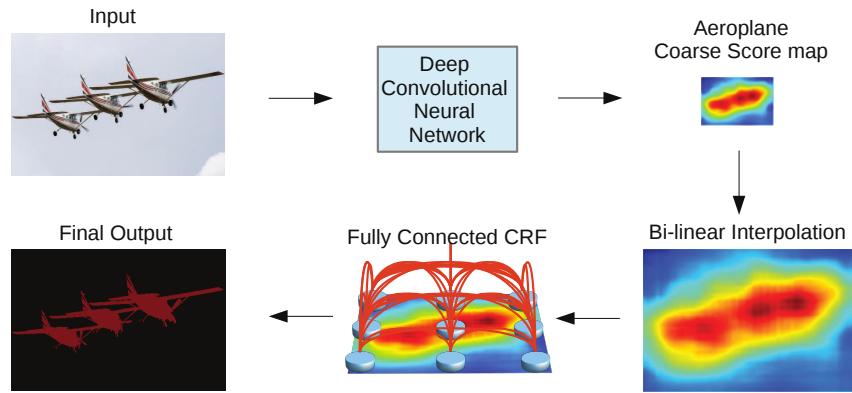


Figure 4.34: A fully connected CRF is added to the output of a CNN, in order to increase the sharpness of the segmentation boundaries. From Figure 3 of [Che+15]. Used with kind permission of Jay Chen.

corresponds to the following conditional model:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \left[\sum_{i \sim j} \psi_{ij}(y_i, y_j) \right] \prod_i p(y_i|\mathbf{x}_i) \quad (4.140)$$

In the sections below, we discuss some applications of this and other CRF models in computer vision.

4.4.2.1 Semantic segmentation

The task of **semantic segmentation** is to assign a label to every pixel in an image. We can easily solve this problem using a CNN with one softmax output node per pixel. However, this may fail to capture long-range dependencies, since convolution is a local operation.

One way to get better results is to feed the output of the CNN into a CRF. Since the CNN already uses convolution, its outputs will usually already be locally smooth, so the benefits from using a CRF with a local grid structure may be quite small. However, we can sometimes get better results if we use a **fully connected CRF**, which has connections between all the pixels. This can capture long range connections which the grid-structured CRF cannot. See Figure 4.34 for an illustration, and [Che+17a] for details.

Unfortunately, exact inference in a fully connected CRF is intractable, but in the case of Gaussian potentials, it is possible to devise an efficient mean field algorithm, as described in [KK11]. Interestingly, [Zhe+15] showed how the mean field update equations can be implemented using a recurrent neural network (see Section 16.3.4), allowing end-to-end training. Alternatively, if we are willing to use a finite number of iterations, we can just “unroll” the computation graph and treat it as a fixed-sized feedforward circuit. The result is a graph-structured neural network, where the topology of the GNN is derived from the graphical model (cf., Section 9.4.10). The advantage of this compared to standard CRF methods is that we can train this entire model end-to-end using standard gradient descent methods; we no longer have to worry about the partition function (see Section 4.4.3), or the lack of convergence that can arise when combining approximate inference with standard CRF learning.

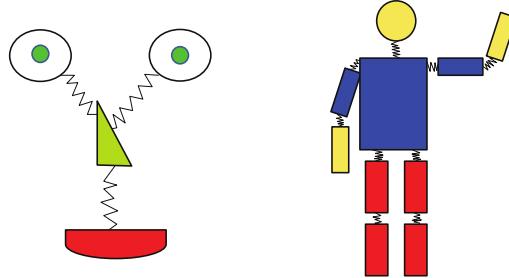


Figure 4.35: Pictorial structures model for a face and body. Each body part corresponds to a node in the CRF whose state space represents the location of that part. The edges (springs) represent pairwise spatial constraints. The local evidence nodes are not shown. Adapted from a figure by Pedro Felzenszwalb.

4.4.2.2 Deformable parts models

Consider the problem of **object detection**, i.e., finding the location(s) of an object of a given class (e.g., a person or a car) in an image. One way to tackle this is to train a binary classifier that takes as input an image patch and specifies if the patch contains the object or not. We can then apply this to every image patch, and return the locations where the classifier has high confidence detections; this is known as a **sliding window detector**, and works quite well for rigid objects such as cars or frontal faces. Such an approach can be made efficient by using convolutional neural networks (CNNs); see Section 16.3.2 for details.

However, such methods can work poorly when there is occlusion, or when the shape is deformable, such as a person's or animal's body, because there is too much variation in the overall appearance. A natural strategy to deal with such problems is break the object into parts, and then to detect each part separately. But we still need to enforce spatial coherence of the parts. This can be done using a pairwise CRF, where node y_i specifies the location of part i in the image (assuming it is present), and where we connect adjacent parts by a potential function that encourages them to be close together. For example, we can use a pairwise potential of the form $\psi(y_i, y_j | \mathbf{x}) = \exp(-d(y_i, y_j))$, where $y_i \in \{1, \dots, K\}$ is the location of part i (a discretization of the 2d image plane), and $d(y_i, y_j)$ is the distance between parts i and j . (We can make this “distance” also depend on the inputs \mathbf{x} if we want, for example we may relax the distance penalty if we detect an edge.) In addition we will have a local evidence term of the form $p(y_i | \mathbf{x})$, which can be any kind of discriminative classifier, such as a CNN, which predicts the distribution over locations for part i given the image \mathbf{x} . The overall model has the form

$$p(\mathbf{y} | \mathbf{x}) = \frac{1}{Z(\mathbf{x})} \left[\prod_i p(y_i | f(\mathbf{x})_i) \right] \left[\prod_{(i,j) \in E} \psi(y_i, y_j | \mathbf{x}) \right] \quad (4.141)$$

where E is the set of edges in the CRF, and $f(\mathbf{x})_i$ is the i 'th output of the CNN.

We can think of this CRF as a series of parts connected by springs, where the energy of the system increases if the parts are moved too far from their expected relative distance. This is illustrated in Figure 4.35. The resulting model is known as a **pictorial structure** [FE73], or **deformable parts**

model [Fel+10]. Furthermore, since this is a conditional model, we can make the spring strengths be image dependent.

We can find the globally optimal joint configuration $\mathbf{y}^* = \operatorname{argmax}_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$ using brute force enumeration in $O(K^T)$ time, where T is the number of nodes and K is the number of states (locations) per node. While T is often small, (e.g., just 10 body parts in Figure 4.35), K is often very large, since there are millions of possible locations in an image. By using tree-structured graphs, exact inference can be done in $O(TK^2)$ time, as we explain in Section 9.3.2. Furthermore, by exploiting the fact that the discrete states are ordinal, inference time can be further reduced to $O(TK)$, as explained in [Fel+10].

Note that by “augmenting” standard deep neural network libraries with a dynamic programming inference “module”, we can represent DPMs as a kind of CNN, as shown in [Gir+15]. The key property is that we can backpropagate gradients through the inference algorithm.

4.4.3 Parameter estimation

In this section, we discuss how to perform maximum likelihood estimation for CRFs. This is a small extension of the MRF case in Section 4.3.9.1.

4.4.3.1 Log-linear potentials

In this section we assume the log potential functions are linear in the parameters, i.e.,

$$\psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) = \exp(\boldsymbol{\theta}_c^\top \phi_c(\mathbf{x}, \mathbf{y}_c)) \quad (4.142)$$

Hence the log likelihood becomes

$$\ell(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_n \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}) = \frac{1}{N} \sum_n \left[\sum_c \boldsymbol{\theta}_c^\top \phi_c(\mathbf{y}_{nc}, \mathbf{x}_n) - \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \right] \quad (4.143)$$

where

$$Z(\mathbf{x}_n; \boldsymbol{\theta}) = \sum_{\mathbf{y}} \exp(\boldsymbol{\theta}^\top \phi(\mathbf{y}, \mathbf{x}_n)) \quad (4.144)$$

is the partition function for example n .

We know from Section 2.4.3 that the derivative of the log partition function yields the expected sufficient statistics, so the gradient of the log likelihood can be written as follows:

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \frac{1}{N} \sum_n \left[\phi_c(\mathbf{y}_{nc}, \mathbf{x}_n) - \frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \right] \quad (4.145)$$

$$= \frac{1}{N} \sum_n [\phi_c(\mathbf{y}_{nc}, \mathbf{x}_n) - \mathbb{E}_{p(\mathbf{y}|\mathbf{x}_n, \boldsymbol{\theta})} [\phi_c(\mathbf{y}, \mathbf{x}_n)]] \quad (4.146)$$

Since the objective is convex, we can use a variety of solvers to find the MLE, such as the stochastic meta descent method of [Vis+06], which is a variant of SGD where the stepsize is adapted automatically.

4.4.3.2 General case

In the general case, a CRF can be written as follows:

$$p(\mathbf{y}|\mathbf{x}; \boldsymbol{\theta}) = \frac{\exp(f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}))}{Z(\mathbf{x}; \boldsymbol{\theta})} = \frac{\exp(f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}))}{\sum_{\mathbf{y}'} \exp(f(\mathbf{x}, \mathbf{y}'; \boldsymbol{\theta}))} \quad (4.147)$$

where $f(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta})$ is a scoring (negative energy) function, where high scores correspond to probable configurations. The gradient of the log likelihood is

$$\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_n, \mathbf{y}_n; \boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} \log Z(\mathbf{x}_n; \boldsymbol{\theta}) \quad (4.148)$$

Computing derivatives of the log partition function is tractable provided we can compute the corresponding expectations, as we discuss in Section 4.3.9.2. Note, however, that we need to compute these derivatives for every training example, which is slower than the MRF case, where the log partition function is a constant independent of the observed data (but dependent on the model parameters).

4.4.4 Other approaches to structured prediction

Many other approaches to structured prediction have been proposed, going beyond CRFs. For example, **max margin Markov networks** [TGK03], and the closely related **structural support vector machine** [Tso+05], can be seen as non-probabilistic alternatives to CRFs. More recently, [BYM17] proposed **structured prediction energy networks**, which are a form of energy based model (Chapter 24), where we predict using an optimization procedure, $\hat{\mathbf{y}}(\mathbf{x}) = \operatorname{argmin} \mathcal{E}(\mathbf{x}, \mathbf{y})$. In addition, it is common to use graph neural networks (Section 16.3.6) and sequence-to-sequence models such as transformers (Section 16.3.5) for this task.

4.5 Comparing directed and undirected PGMs

In this section, we compare DPGMs and UPGMs in terms of their modeling power, we discuss how to convert from one to the other, and we present a unified representation.

4.5.1 CI properties

Which model has more “expressive power”, a DPGM or a UPGM? To formalize the question, recall from Section 4.2.4 that G is an I-map of a distribution p if $I(G) \subseteq I(p)$, meaning that all the CI statements encoded by the graph G are true of the distribution p . Now define G to be **perfect map** of p if $I(G) = I(p)$, in other words, the graph can represent all (and only) the CI properties of the distribution. It turns out that DPGMs and UPGMs are perfect maps for different sets of distributions (see Figure 4.36). In this sense, neither is more powerful than the other as a representation language.

As an example of some CI relationships that can be perfectly modeled by a DPGM but not a UPGM, consider a v-structure $A \rightarrow C \leftarrow B$. This asserts that $A \perp B$, and $A \not\perp B|C$. If we drop the arrows, we get $A - C - B$, which asserts $A \perp B|C$ and $A \not\perp B$, which is not consistent with the independence statements encoded by the DPGM. In fact, there is no UPGM that can precisely

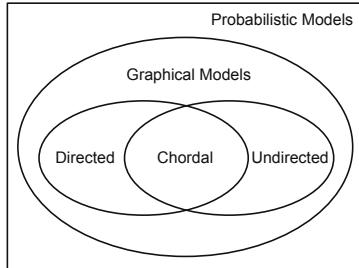


Figure 4.36: DPGMs and UPGMs can perfectly represent different sets of distributions. Some distributions can be perfectly represented by either DPGM's or UPGMs; the corresponding graph must be chordal.

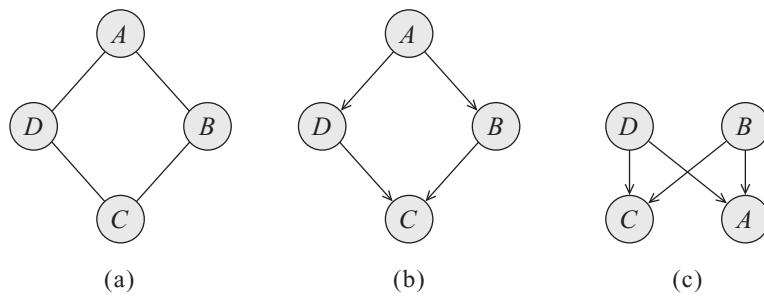


Figure 4.37: A UPGM and two failed attempts to represent it as a DPGM. From Figure 3.10 of [KF09a]. Used with kind permission of Daphne Koller.

represent all and only the two CI statements encoded by a v-structure. In general, CI properties in UPGMs are monotonic, in the following sense: if $A \perp B|C$, then $A \perp B|(C \cup D)$. But in DPGMs, CI properties can be non-monotonic, since conditioning on extra variables can eliminate conditional independencies due to explaining away.

As an example of some CI relationships that can be perfectly modeled by a UPGM but not a DPGM, consider the 4-cycle shown in Figure 4.37(a). One attempt to model this with a DPGM is shown in Figure 4.37(b). This correctly asserts that $A \perp C|B, D$. However, it incorrectly asserts that $B \perp D|A$. Figure 4.37(c) is another incorrect DPGM: it correctly encodes $A \perp C|B, D$, but incorrectly encodes $B \perp D$. In fact there is no DPGM that can precisely represent all and only the CI statements encoded by this UPGM.

Some distributions can be perfectly modeled by either a DPGM or a UPGM; the resulting graphs are called **decomposable** or **chordal**. Roughly speaking, this means the following: if we collapse together all the variables in each maximal clique, to make “mega-variables”, the resulting graph will be a tree. Of course, if the graph is already a tree (which includes chains as a special case), it will already be chordal.

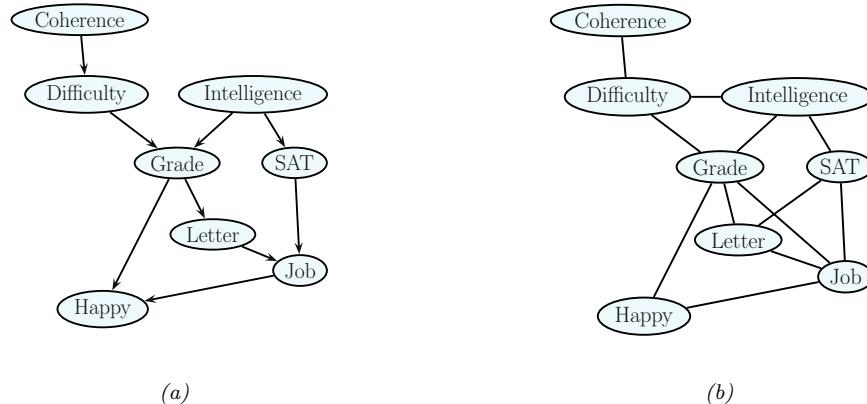


Figure 4.38: Left: the full student DPGM. Right: the equivalent UPGM. We add moralization arcs $D-I$, $G-J$, and $L-S$. Adapted from Figure 9.8 of [KF09a].

4.5.2 Converting between a directed and undirected model

Although DPGMs and UPGMs are not in general equivalent, if we are willing to allow the graph to encode fewer CI properties than may strictly hold, then we can safely convert one to the other, as we explain below.

4.5.2.1 Converting a DPGM to a UPGM

We can easily convert a DPGM to a UPGM as follows. First, any “unmarried” parents that share a child must get “married”, by adding an edge between them; this process is known as **moralization**. Then we can drop the arrows, resulting in an undirected graph. The reason we need to do this is to ensure that the CI properties of the UGM match those of the DGM, as explained in Section 4.3.6.2. It also ensures there is a clique that can “store” the CPDs of each family.

Let us consider an example from [KF09a]. We will use the (full version of the student network shown in Figure 4.38(a). The corresponding joint has the following form:

$$P(C, D, I, G, S, L, J, H) \quad (4.149)$$

$$= P(C)P(D|C)P(I)P(G|I, D)P(S|I)P(L|G)P(J|L, S)P(H|G, J) \quad (4.150)$$

Next, we define a potential or factor for every CPD, yielding

$$p(C, D, I, G, S, L, J, H) = \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, I, D) \quad (4.151)$$

$$\psi_S(S, I)\psi_L(L, G)\psi_J(J, L, S)\psi_H(H, G, J) \quad (4.152)$$

All the potentials are **locally normalized**, since they are CPDs, and hence there is no need for a global normalization constant, so $Z = 1$. The corresponding undirected graph is shown in Figure 4.38(b). We see that we have added D-I, G-J, and L-S moralization edges.¹¹

¹¹ We will see this example again in Section 9.5, where we use it to illustrate the variable elimination inference algorithm.

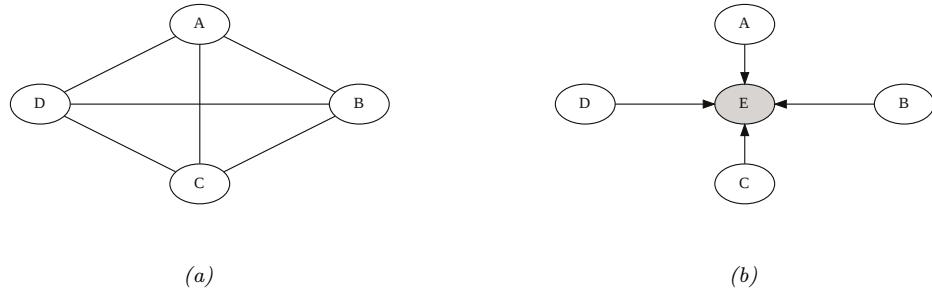


Figure 4.39: (a) An undirected graphical model. (b) A directed equivalent, obtained by adding a dummy observed child node.

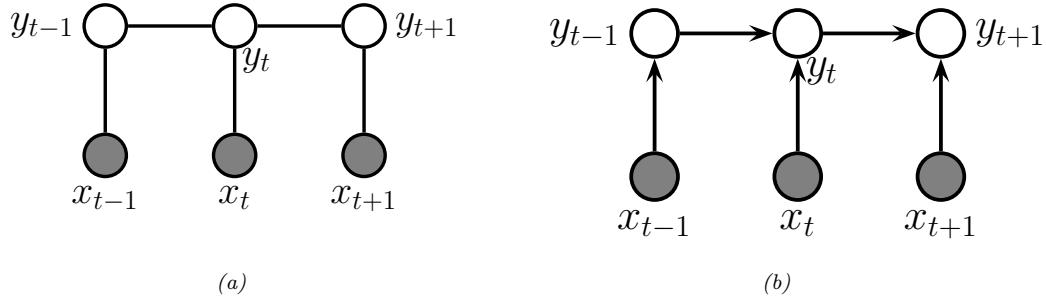


Figure 4.40: Two discriminative models for sequential data. (a) An undirected model (CRF). (b) A directed model (MEMM).

4.5.2.2 Converting a UPGM to a DPGM

To convert a UPGM to a DPGM, we proceed as follows. For each potential function $\psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c)$, we create a “dummy node”, call it Y_c , which is “clamped” to a special observed state, call it y_c^* . We then define $p(Y_c = y_c^* | \mathbf{x}_c) = \psi_c(\mathbf{x}_c; \boldsymbol{\theta}_c)$. This “local evidence” CPD encodes the same factor as in the DGM. The overall joint has the form $p_{\text{undir}}(\mathbf{x}) \propto p_{\text{dir}}(\mathbf{x}, \mathbf{y}^*)$.

As an example, consider the UPGM in Figure 4.39(a), which defines the joint $p(A, B, C, D) = \psi(A, B, C, D)/Z$. We can represent this as a DPGM by adding a dummy E node, which is a child of all the other nodes. We set $E = 1$ and define the CPD $p(E = 1 | A, B, C, D) \propto \psi(A, B, C, D)$. By conditioning on this observed child, all the parents become dependent, as in the UGM.

4.5.3 Conditional directed vs undirected PGMs and the label bias problem

Directed and undirected models behave somewhat differently in the conditional (discriminative) setting. As an example of this, let us compare the 1d undirected CRF in Figure 4.40a with the directed Markov chain in Figure 4.40b. (This latter model is called a maximum entropy Markov model

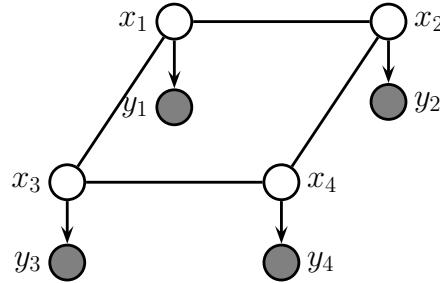


Figure 4.41: A grid-structured MRF with hidden nodes x_i and local evidence nodes y_i . The prior $p(\mathbf{x})$ is an undirected Ising model, and the likelihood $p(\mathbf{y}|\mathbf{x}) = \prod_i p(y_i|x_i)$ is a directed fully factored model.

(MEMM), which is a reference to the connection with maxent models discussed in Section 4.3.4.) The MEMM suffers from a subtle problem compared to the CRF known (rather obscurely) as the **label bias** problem [LMP01]. The problem is that local features at time t do not influence states prior to time t . That is, $y_{t-1} \perp \mathbf{x}_t | y_t$, thus blocking information flow backwards in time.

To understand what this means in practice, consider the part of speech tagging task which we discussed in Section 4.4.1.1. Suppose we see the word “banks”; this could be a verb (as in “he banks at Chase”), or a noun (as in “the river banks were overflowing”). Locally the part of speech tag for the word is ambiguous. However, suppose that later in the sentence, we see the word “fishing”; this gives us enough context to infer that the sense of “banks” is “river banks” and not “financial banks”. However, in an MEMM the “fishing” evidence will not flow backwards, so we will not be able to infer the correct label for “banks”. The CRF does not have this problem.

The label bias problem in MEMMs occurs because directed models are **locally normalized**, meaning each CPD sums to 1. By contrast, MRFs and CRFs are **globally normalized**, which means that local factors do not need to sum to 1, since the partition function Z , which sums over all joint configurations, will ensure the model defines a valid distribution.

However, this solution comes at a price: in a CRF, we do not get a valid probability distribution over $\mathbf{y}_{1:T}$ until we have seen the whole sentence, since only then can we normalize over all configurations. Consequently, CRFs are not as useful as directed probabilistic graphical models (DPGM) for online or real-time inference. Furthermore, the fact that Z is a function of all the parameters makes CRFs less modular and much slower to train than DPGM’s, as we discuss in Section 4.4.3.

4.5.4 Combining directed and undirected graphs

We can also define graphical models that contain directed and undirected edges. We discuss a few examples below.

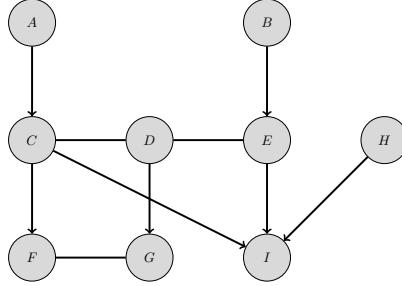


Figure 4.42: A partially directed acyclic graph (PDAG). The chain components are $\{A\}$, $\{B\}$, $\{C, D, E\}$, $\{F, G\}$, $\{H\}$, and $\{I\}$. Adapted from Figure 4.15 of [KF09a].

4.5.4.1 Chain graphs

A **chain graph** is a PGM which may have both directed and undirected edges, but without any directed cycles. A simple example is shown in Figure 4.41, which defines the following joint model:

$$p(\mathbf{x}_{1:D}, \mathbf{y}_{1:D}) = p(\mathbf{x}_{1:D})p(\mathbf{y}_{1:D}|\mathbf{x}_{1:D}) = \left[\frac{1}{Z} \prod_{i \sim j} \psi_{ij}(x_i, x_j) \right] \left[\prod_{i=1}^D p(y_i|x_i) \right] \quad (4.153)$$

In this example, the prior $p(\mathbf{x})$ is specified by a UPGM, and the likelihood $p(\mathbf{y}|\mathbf{x})$ is specified as a fully factorized DPGM.

More generally, a chain graph can be defined in terms of a **partially directed acyclic graph (PDAG)**. This is a graph which can be decomposed into a directed graph of **chain components**, where the nodes within each chain component are connected with each other only with undirected edges. See Figure 4.42 for an example.

We can use a PDAG to define a joint distribution using $\prod_i p(C_i|\text{pa}(C_i))$, where each C_i is a chain component, and each CPD is a conditional random field. For example, referring to Figure 4.42, we have

$$p(A, B, \dots, I) = p(A)p(B)p(C, D, E|A, B)p(F, G|C, D)p(H)p(I|C, E, H) \quad (4.154)$$

$$p(C, D, E|A, B) = \frac{1}{Z(A, B)} \phi(A, C)\phi(B, E)\phi(C, D)\phi(D, E) \quad (4.155)$$

$$p(F, G|C, D) = \frac{1}{Z(C, D)} \phi(F, C)\phi(G, D)\phi(F, G) \quad (4.156)$$

For more details, see e.g., [KF09a, Sec 4.6.2].

4.5.4.2 Acyclic directed mixed graphs

One can show [Pea09b, p51] that every latent variable DPGM can be rewritten in a way such that every latent variable is a root node with exactly two observed children. This is called the **projection** of the latent variable PGM, and is observationally indistinguishable from the original model.

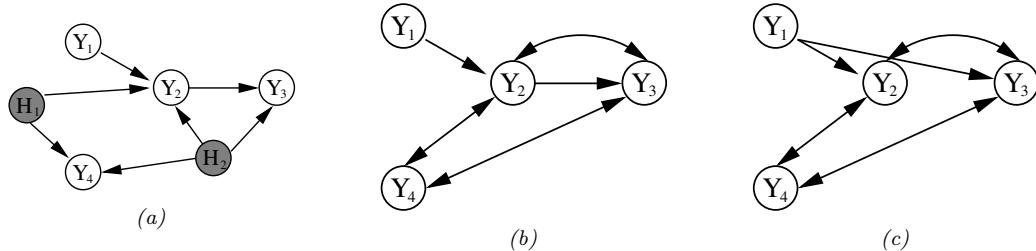


Figure 4.43: (a) A DAG with two hidden variables (shaded). (b) The corresponding ADMG. The bidirected edges reflect correlation due to the hidden variable. (c) A Markov equivalent ADMG. From Figure 3 of [SG09]. Used with kind permission of Ricardo Silva.

Each such latent variable root node induces a dependence between its two children. We can represent this with a directed arc. The resulting graph is called an **acyclic directed mixed graph** or **ADMG**. See Figure 4.43 for an example. (A **mixed graph** is one with undirected, unidirected, and bidirected edges.)

One can determine CI properties of ADMGs using a technique called **m-separation** [Ric03]. This is equivalent to d-separation in a graph where every bidirected edge $Y_i \leftrightarrow Y_j$ is replaced by $Y_i \leftarrow X_{ij} \rightarrow Y_j$, where X_{ij} is a hidden variable for that edge.

The most common example of ADMGs is when everything is linear-Gaussian. This is known as a structural equation model and is discussed in Section 4.7.2.

4.5.5 Comparing directed and undirected Gaussian PGMs

In this section, we compare directed and undirected Gaussian graphical models. In Section 4.2.3, we saw that directed GGMs correspond to sparse regression matrices. In Section 4.3.5, we saw that undirected GGMs correspond to sparse precision matrices.

The advantage of the DAG formulation is that we can make the regression weights \mathbf{W} , and hence Σ , be conditional on covariate information [Pou04], without worrying about positive definite constraints. The disadvantage of the DAG formulation is its dependence on the order, although in certain domains, such as time series, there is already a natural ordering of the variables.

It is actually possible to combine both directed and undirected representations, resulting in a model known as a (Gaussian) **chain graph**. For example, consider a discrete-time, second-order Markov chain in which the observations are continuous, $\mathbf{x}_t \in \mathbb{R}^D$. The transition function can be represented as a (vector-valued) linear-Gaussian CPD:

$$p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t | \mathbf{A}_1 \mathbf{x}_{t-1} + \mathbf{A}_2 \mathbf{x}_{t-2}, \boldsymbol{\Sigma}) \quad (4.157)$$

This is called a **vector autoregressive** or **VAR** process of order 2. Such models are widely used in econometrics for time series forecasting.

The time series aspect is most naturally modeled using a DPGM. However, if $\boldsymbol{\Sigma}^{-1}$ is sparse, then the correlation amongst the components within a time slice is most naturally modeled using a UPGM.

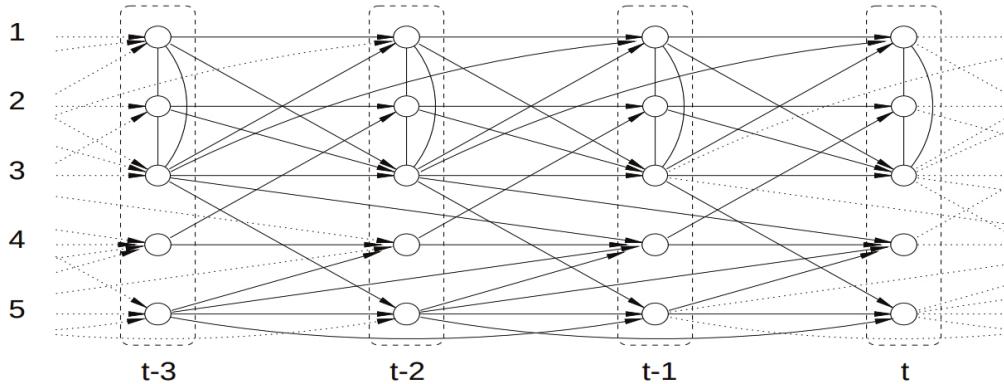


Figure 4.44: A VAR(2) process represented as a dynamic chain graph. From [DE00]. Used with kind permission of Rainer Dahlhaus.

For example, suppose we have

$$\mathbf{A}_1 = \begin{pmatrix} \frac{3}{5} & 0 & \frac{1}{5} & 0 & 0 \\ 0 & \frac{3}{5} & 0 & -\frac{1}{5} & 0 \\ \frac{2}{5} & \frac{1}{3} & \frac{3}{5} & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{5} \\ 0 & 0 & \frac{1}{5} & 0 & \frac{3}{5} \end{pmatrix}, \quad \mathbf{A}_2 = \begin{pmatrix} 0 & 0 & -\frac{1}{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{5} \end{pmatrix} \quad (4.158)$$

and

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & 0 & 0 \\ \frac{1}{2} & 1 & -\frac{1}{3} & 0 & 0 \\ \frac{1}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} 2.13 & -1.47 & -1.2 & 0 & 0 \\ -1.47 & 2.13 & 1.2 & 0 & 0 \\ -1.2 & 1.2 & 1.8 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.159)$$

The resulting graphical model is illustrated in Figure 4.44. Zeros in the transition matrices \mathbf{A}_1 and \mathbf{A}_2 correspond to absent directed arcs from \mathbf{x}_{t-1} and \mathbf{x}_{t-2} into \mathbf{x}_t . Zeros in the precision matrix $\boldsymbol{\Sigma}^{-1}$ correspond to absent undirected arcs between nodes in \mathbf{x}_t .

4.5.5.1 Covariance graphs

Sometimes we have a sparse covariance matrix rather than a sparse precision matrix. This can be represented using a **bi-directed graph**, where each edge has arrows in both directions, as in Figure 4.45(a). Here nodes that are not connected are unconditionally independent. For example in Figure 4.45(a) we see that $Y_1 \perp Y_3$. In the Gaussian case, this means $\Sigma_{1,3} = \Sigma_{3,1} = 0$. (A graph representing a sparse covariance matrix is called a **covariance graph**, see e.g., [Pen13]). By contrast, if this were an undirected model, we would have that $Y_1 \perp Y_3 | Y_2$, and $\Lambda_{1,3} = \Lambda_{3,1} = 0$, where $\Lambda = \boldsymbol{\Sigma}^{-1}$.

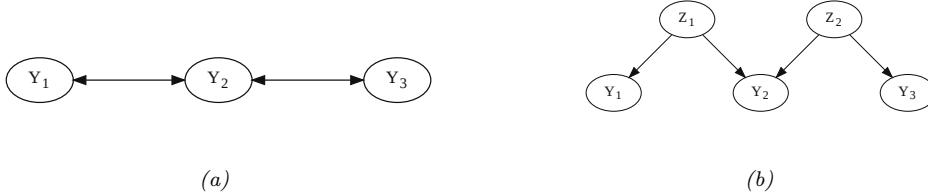


Figure 4.45: (a) A bi-directed graph. (b) The equivalent DAG. Here the z nodes are latent confounders. Adapted from Figures 5.12–5.13 of [Cho11].

A bidirected graph can be converted to a DAG with latent variables, where each bidirected edge is replaced with a hidden variable representing a hidden common cause, or **confounder**, as illustrated in Figure 4.45(b). The relevant CI properties can then be determined using d-separation.

4.6 PGM extensions

In this section, we discuss some extensions of the basic PGM framework.

4.6.1 Factor graphs

A **factor graph** [KFL01; Loe04] is a graphical representation that unifies directed and undirected models. They come in two main “flavors”. The original version uses a bipartite graph, where we have nodes for random variables and nodes for factors, as we discuss in Section 4.6.1.1. An alternative form, known as a **Forney factor graphs** [For01] just has nodes for factors, and the variables are associated with edges, as we explain in Section 4.6.1.2.

4.6.1.1 Bipartite factor graphs

A **factor graph** is an undirected bipartite graph with two kinds of nodes. Round nodes represent variables, square nodes represent factors, and there is an edge from each variable to every factor that mentions it. For example, consider the MRF in Figure 4.46(a). If we assume one potential per maximal clique, we get the factor graph in Figure 4.46(b), which represents the function

$$f(x_1, x_2, x_3, x_4) = f_{124}(x_1, x_2, x_4)f_{234}(x_2, x_3, x_4) \quad (4.160)$$

We can represent this in a topologically equivalent way as in Figure 4.46(c).

One advantage of factor graphs over UPGM diagrams is that they are more fine-grained. For example, suppose we associate one potential per edge, rather than per clique. In this case, we get the factor graph in Figure 4.46(d), which represents the function

$$f(x_1, x_2, x_3, x_4) = f_{14}(x_1, x_4)f_{12}(x_1, x_2)f_{34}(x_3, x_4)f_{23}(x_2, x_3)f_{24}(x_2, x_4) \quad (4.161)$$

We can also convert a DPGM to a factor graph: just create one factor per CPD, and connect that factor to all the variables that use that CPD. For example, Figure 4.47 represents the following

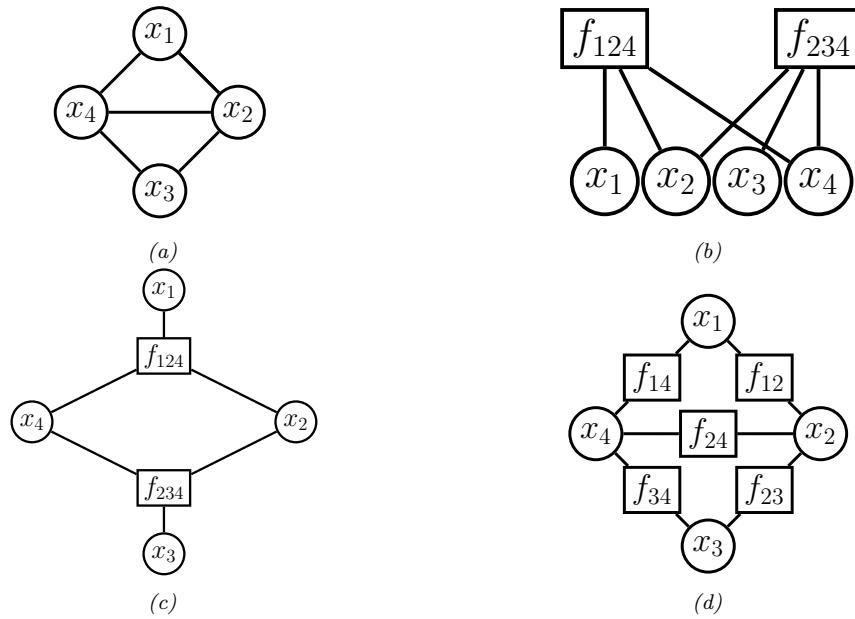


Figure 4.46: (a) A simple UPGM. (b) A factor graph representation assuming one potential per maximal clique. (c) Same as (b), but graph is visualized differently. (d) A factor graph representation assuming one potential per edge.

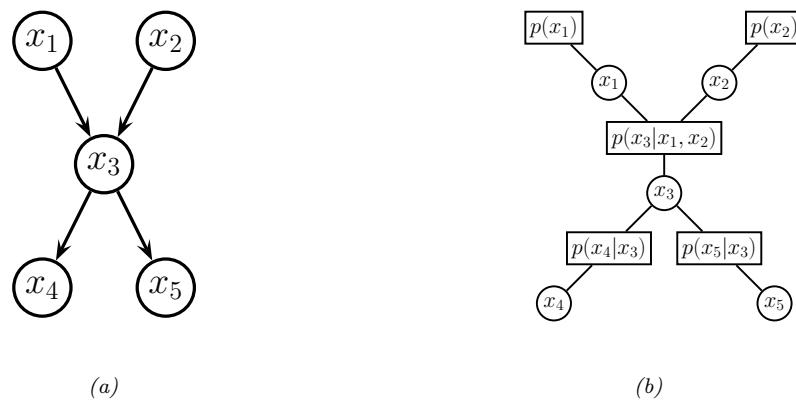


Figure 4.47: (a) A simple DPGM. (b) Its corresponding factor graph.

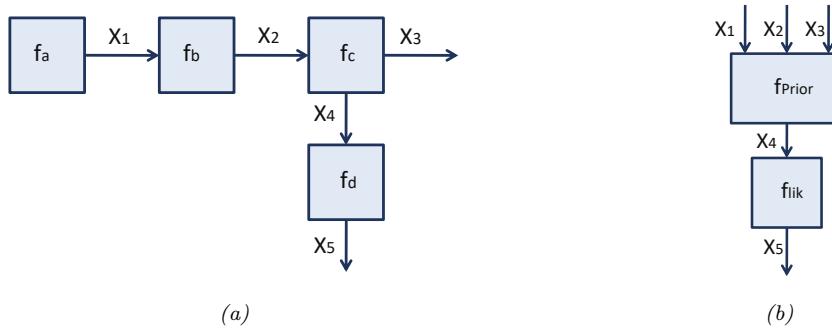


Figure 4.48: A Forney factor graph. (a) Directed version. (b) Hierarchical version.

factorization:

$$f(x_1, x_2, x_3, x_4, x_5) = f_1(x_1)f_2(x_2)f_{123}(x_1, x_2, x_3)f_{34}(x_3, x_4)f_{35}(x_3, x_5) \quad (4.162)$$

where we define $f_{123}(x_1, x_2, x_3) = p(x_3|x_1, x_2)$, etc. If each node has at most one parent (and hence the graph is a chain or simple tree), then there will be one factor per edge (root nodes can have their prior CPDs absorbed into their children's factors). Such models are equivalent to pairwise MRFs.

4.6.1.2 Forney factor graphs

A **Forney factor graph (FFG)**, also called a **normal factor graph**, is a graph in which nodes represent factors, and edges represent variables [For01; Loe04; Loe+07; CLV19]. This is more similar to standard neural network diagrams, and electrical engineering diagrams, where signals (represented as electronic pulses, or tensors, or probability distributions) propagate along wires and are modified by functions represented as nodes.

For example, consider the following factorized function:

$$f(x_1, \dots, x_5) = f_a(x_1)f_b(x_1, x_2)f_c(x_2, x_3, x_4)f_d(x_4, x_5) \quad (4.163)$$

We can visualize this as an FFG as in Figure 4.48a. The edge labeled x_3 is called a **half-edge**, since it is only connected to one node; this is because x_3 only participates in one factor. (Similarly for x_5 .) The directionality associated with the edges is a useful mnemonic device if there is a natural order in which the variables are generated. In addition, associating directions with each edge allows us to uniquely name “messages” that are sent along each edge, which will prove useful when we discuss inference algorithms in Section 9.3.

In addition to being more similar to neural network diagrams, FFGs have the advantage over bipartite FGs in that they support hierarchical (compositional) construction, in which complex dependency structure between variables can be represented as a blackbox, with the input/output interface being represented by edges corresponding to the variables exposed by the blackbox. See Figure 4.48b for an example, which represents the function

$$f(x_1, \dots, x_5) = f_{\text{prior}}(x_1, x_2, x_3, x_4)f_{\text{lik}}(x_4, x_5) \quad (4.164)$$

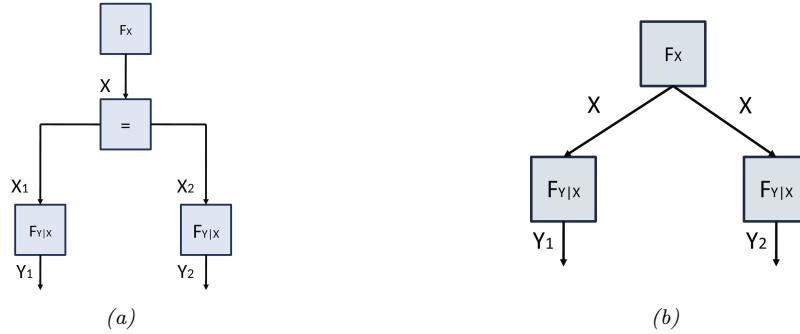


Figure 4.49: An FFG with an equality constraint node (left) and its corresponding simplified form (right).

The factor f_{prior} represents a (potentially complex) joint distribution $p(x_1, x_2, x_3, x_4)$, and the factor f_{lik} represents the likelihood term $p(x_5|x_4)$. Such models are widely used to build error-correcting codes (see Section 9.4.8).

To allow for variables to participate in more than 2 factors, equality constraint nodes are introduced, as illustrated in Figure 4.49(a). Formally, this is a factor defined as follows:

$$f_=(x, x_1, x_2) = \delta(x - x_1)\delta(x - x_2) \quad (4.165)$$

where $\delta(u)$ is a Dirac delta if u is continuous, and a Kronecker delta if u is discrete. The effect of this factor is to ensure all the variables connected to the factor have the same value; intuitively, this factor acts like a “wire splitter”. Thus the function represented in Figure 4.49(a) is equivalent to the following:

$$f(x, y_1, y_2) = f_x(x)f_{y|x}(y_1, x)f_{y|x}(y_2, x) \quad (4.166)$$

This simplified form is represented in Figure 4.49(b), where we reuse the x variable across multiple edges. We have chosen the edge orientations to reflect our interpretation of the factors $f_{y|x}(y, x)$ as likelihood terms, $p(y|x)$. We have also chosen to reuse the same $f_{y|x}$ factor for both y variables; this is an example of **parameter tying**.

4.6.2 Probabilistic circuits

A **probabilistic circuit** is a kind of graphical model that supports efficient exact inference. It includes **arithmetic circuits** [Dar03; Dar09], **sum-product networks** (SPNs) [PD11; SCPD22], and other kinds of model.

Here we briefly describe SPNs. An SPN is a probabilistic model, based on a directed tree-structured graph, in which terminal nodes represent univariate probability distributions and non-terminal nodes represent convex combinations (weighted sums) and products of probability functions. SPNs are similar to deep mixture models, in which we combine together dimensions. SPNs leverage context-specific independence to reduce the complexity of exact inference to time that is proportional to the number of links in the graph, as opposed to the treewidth of the graph (see Section 9.5.2).

SPNs are particularly useful for tasks such as missing data imputation of tabular data (see e.g., [Cla20; Ver+19]). A recent extension of SPNs, known as **einsum networks**, is proposed in [Peh+20] (see Section 9.7.1 for details on the connection between einstein summation and PGM inference).

4.6.3 Directed relational PGMs

A Bayesian network defines a joint probability distribution over a fixed number of random variables. By using plate notation (Section 4.2.8), we can define models with certain kinds of repetitive structure, and tied parameters, but many models are not expressible in this way. For example, it is not possible to represent even a simple HMM using plate notation (see Figure 29.12). Various notational extensions of plates have been proposed to handle repeated structure (see e.g., [HMK04; Die10]) but have not been widely adopted. The problem becomes worse when we have more complex domains, involving multiple objects which interact via multiple relationships.¹² Such models are called **relational probability models** or **RPMs**. In this section, we focus on directed RPMs; see Section 4.6.4 for the undirected case.

As in first order logic, RPMs have constant symbols (representing objects), function symbols (mapping one set of constants to another), and predicate symbols (representing relations between objects). We will assume that each function has a **type signature**. To illustrate this, consider an example from [RN19, Sec 15.1], which concerns online book reviews on sites such as Amazon. Suppose there are two types of objects, Book and Customer, and the following functions and predicates:

$$\text{Honest} : \text{Customer} \rightarrow \{\text{True}, \text{False}\} \quad (4.167)$$

$$\text{Kindess} : \text{Customer} \rightarrow \{1, 2, 3, 4, 5\} \quad (4.168)$$

$$\text{Quality} : \text{Book} \rightarrow \{1, 2, 3, 4, 5\} \quad (4.169)$$

$$\text{Recommendation} : \text{Customer} \times \text{Book} \rightarrow \{1, 2, 3, 4, 5\} \quad (4.170)$$

The constant symbols refer to specific objects. To keep things simple, we assume there are two books, B_1 and B_2 , and two customers, C_1 and C_2 . The **basic random variables** are obtained by instantiating each function with each possible combination of objects to create a set of **ground terms**. In this example, these variables are $H(C_1)$, $Q(B_1)$, $R(C_1, B_2)$, etc. (We use the abbreviations H , K , Q and R for the functions Honest, Kindness, Quality, and Recommendation.¹³)

We now need to specify the (conditional) distribution over these random variables. We define these distributions in terms of the generic indexed form of the variables, rather than the specific ground form. For example, we may use the following priors for the root nodes (variables with no parents):

$$H(c) \sim \text{Cat}(0.99, 0.01) \quad (4.171)$$

$$K(c) \sim \text{Cat}(0.1, 0.1, 0.2, 0.3, 0.3) \quad (4.172)$$

$$Q(b) \sim \text{Cat}(0.05, 0.2, 0.4, 0.2, 0.15) \quad (4.173)$$

For the recommendation nodes, we need to define a conditional distribution of the form

$$R(c, b) \sim \text{RecCPD}(H(c), K(c), Q(b)) \quad (4.174)$$

12. See e.g., this blog post from Rob Zinkov: <https://www.zinkov.com/posts/2013-07-28-stop-using-plates>.

13. A unary function of an object that returns a basic type, such as Boolean or an integer, is often called an **attribute** of that object.

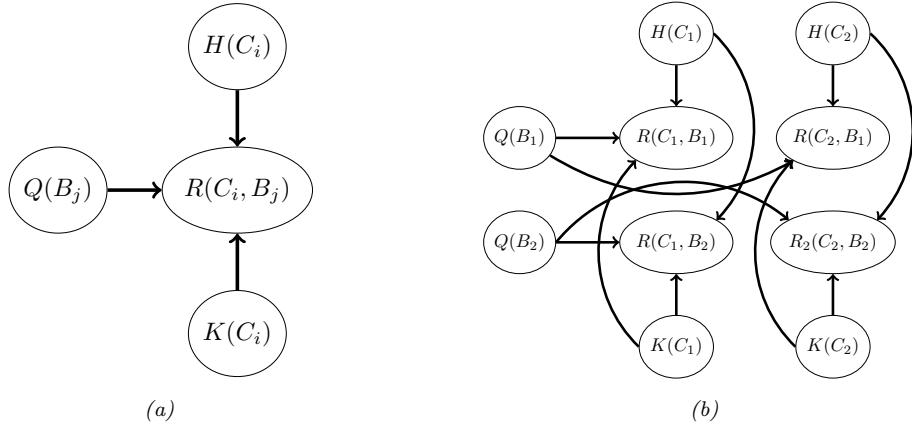


Figure 4.50: RPM for the book review domain. (a) Template for a generic customer C_i and book B_j pair. R is rating, Q is quality, H is honesty, and K is kindness. (b) Unrolled model for 2 books and 2 customers.

where RecCPD is the conditional probability distribution (CPD) for the recommendation node. If represented as a conditional probability table (CPT), this has $2 \times 5 \times 5 = 50$ rows, each with 5 entries. This table can encode our assumptions about what kind of ratings a book receives based on the quality of the book, but also properties of the reviewer, such as their honest and kindness. (More sophisticated models of human raters in the context of crowd-sourced data collection can be found in e.g., [LRC19].)

We can convert the above formulae into a graphical model “**template**”, as shown in Figure 4.50a. Given a set of objects, we can “**unroll**” the template to create a “**ground network**”, as shown in Figure 4.50b. There are $C \times B + 2C + B$ random variables, with a corresponding joint state space (set of **possible worlds**) of size $2^C 5^{C+B+BC}$, which can get quite large. However, if we are only interested in answering specific queries, we can dynamically unroll small pieces of the network that are relevant to that query [GC90; Bre92].

Let us assume that only a subset of the $R(c, b)$ entries are observed, and we would like to predict the missing entries of this matrix. This is essentially a simplified **recommender system**. (Unfortunately it ignores key aspects of the problem, such as the content/topic of the books, and the interests/preferences of the customers.) We can use standard probabilistic inference methods for graphical models (which we discuss in Chapter 9) to solve this problem.

Things get more interesting when we don’t know which objects are being referred to. For example, customer C_1 might write a review of a book called “Probabilistic Machine Learning”, but do they mean edition 1 (B_1) or edition 2 (B_2)? To handle this kind of **relational uncertainty**, we can add all possible referents as parents to each relation. This is illustrated in Figure 4.51, where now $Q(B_1)$ and $Q(B_2)$ are both parents of $R(C_1, B_1)$. This is necessary because their review score might either depend on $Q(B_1)$ or $Q(B_2)$, depending on which edition they are writing about. To disambiguate this, we create a new variable, $L(C_i)$, which specifies which version number of each book customer i is referring to. The new CPD for the recommendation node, $p(R(c, b)|H(c), K(c), Q(1 : B), L(c))$,

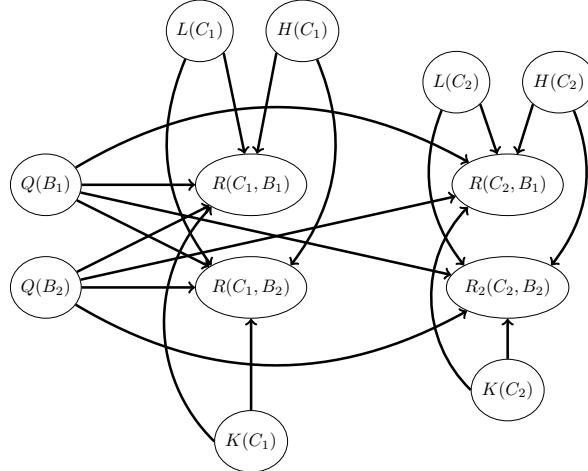


Figure 4.51: An extension of the book review RPM to handle identity uncertainty about which book a given customer is actually reviewing. The $R(c, b)$ node now depends on all books, since we don't know which one is being referred to. We can select one of these parents based on the mapping specified by the user's library, $L(c)$.

has the form

$$R(c, b) \sim \text{RecCPT}(H(c), K(c), Q(b')) \text{ where } b' = L(c) \quad (4.175)$$

This CPD acts like a **multiplexer**, where the $L(c)$ node specifies which of the parents $Q(1 : B)$ to actually use.

Although the above problem may seem contrived, **identity uncertainty** is a widespread problem in many areas, such as citation analysis, credit card histories, and object tracking (see Section 4.6.5). In particular, the problem of **entity resolution** or **record linkage** — which refers to the task of mapping particular strings (such as names) to particular objects (such as people) — is a whole field of research (see e.g., https://en.wikipedia.org/wiki/Record_linkage for an overview and [SHF15] for a Bayesian approach).

4.6.4 Undirected relational PGMs

We can create **relational UGMs** in a manner which is analogous to relational DGMs (Section 4.6.3). This is particularly useful in the discriminative setting, for the same reasons that undirected CRFs are preferable to conditional DGMs (see Section 4.4).

4.6.4.1 Collective classification

As an example of a relational UGM, suppose we are interested in the problem of classifying web pages of a university into types (e.g., student, professor, admin, etc.) Obviously we can do this based on the contents of the page (e.g., words, pictures, layout, etc.) However, we might also suppose there is information in the hyper-link structure itself. For example, it might be likely for students to

Fr(A,A)	Fr(B,B)	Fr(B,A)	Fr(A,B)	Sm(A)	Sm(B)	Ca(A)	Ca(B)
1	1	0	1	1	1	1	1
1	1	0	1	1	0	0	0
1	1	0	1	1	1	0	1

Table 4.5: Some possible joint instantiations of the 8 variables in the smoking example.

cite professors, and professors to cite other professors, but there may be no links between admin pages and students/professors. When faced with a web page whose label is ambiguous, we can bias our estimate based on the estimated labels of its neighbors, as in a CRF. This process is known as **collective classification** (see e.g., [Sen+08]). To specify the CRF structure for a web-graph of arbitrary size and shape, we just specify a template graph and potential functions, and then unroll the template appropriately to match the topology of the web, making use of parameter tying.

4.6.4.2 Markov logic networks

One particularly popular way of specifying relational UGMs is to use **first-order logic** rather than a graphical description of the template. The result is known as a **Markov logic network** [RD06; Dom+06; DL09].

For example, consider the sentences “Smoking causes cancer” and “If two people are friends, and one smokes, then so does the other”. We can write these sentences in first-order logic as follows:

$$\forall x. Sm(x) \implies Ca(x) \quad (4.176)$$

$$\forall x. \forall y. Fr(x, y) \wedge Sm(x) \implies Sm(y) \quad (4.177)$$

where Sm and Ca are predicates, and Fr is a relation.

It is convenient to write all formulas in **conjunctive normal form** (CNF), also known as **clausal form**. In this case, we get

$$\neg Sm(x) \vee Ca(x) \quad (4.178)$$

$$\neg Fr(x, y) \vee \neg Sm(x) \vee Sm(y) \quad (4.179)$$

The first clause can be read as “Either x does not smoke or he has cancer”, which is logically equivalent to Equation (4.176). (Note that in a clause, any unbound variable, such as x , is assumed to be universally quantified.)

Suppose there are just two objects (people) in the world, Anna and Bob, which we will denote by **constant symbols** A and B . We can then create 8 binary random variables $Sm(x)$, $Ca(x)$, and $Fr(x, y)$ for $x, y \in \{A, B\}$. This defines 2^8 **possible worlds**, some of which are shown in Table 4.5.¹⁴

Our goal is to define a probability distribution over these joint assignments. We can do this by creating a UGM with these variables, and adding a potential function to capture each logical rule or

14. Note that we have not encoded the fact that Fr is a symmetric relation, so $Fr(A, B)$ and $Fr(B, A)$ might have different values. Similarly, we have the “degenerate” nodes $Fr(A)$ and $Fr(B)$, since we did not enforce $x \neq y$ in Equation (4.177). (If we add such constraints, then the model compiler, which generates the ground network, should avoid creating redundant nodes.)

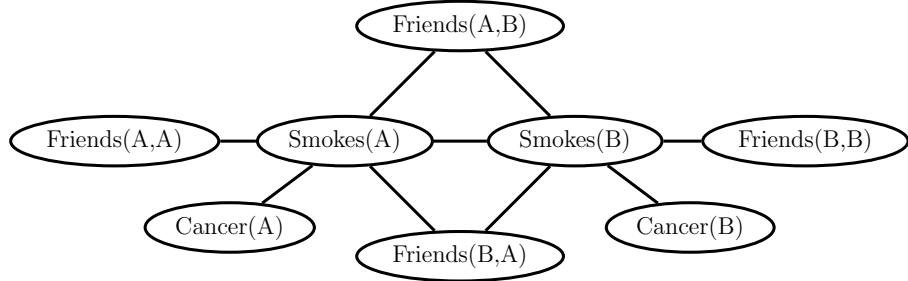


Figure 4.52: An example of a ground Markov logic network represented as a pairwise MRF for 2 people. Adapted from Figure 2.1 from [DL09]. Used with kind permission of Pedro Domingos.

constraint. For example, we can encode the rule $\neg Sm(x) \vee Ca(x)$ by creating a potential function $\Psi(Sm(x), Ca(x))$, where we define

$$\Psi(Sm(x), Ca(x)) = \begin{cases} 1 & \text{if } \neg Sm(x) \vee Ca(x) = T \\ 0 & \text{if } \neg Sm(x) \vee Ca(x) = F \end{cases} \quad (4.180)$$

The result is the UGM in Figure 4.52.

The above approach will assign non-zero probability to all logically valid worlds. However, logical rules may not always be true. For example, smoking does not always cause cancer. We can relax the hard constraints by using non-zero potential functions. In particular, we can associate a weight with each rule, and thus get potentials such as

$$\Psi(Sm(x), Ca(x)) = \begin{cases} e^w & \text{if } \neg Sm(x) \vee Ca(x) = T \\ e^0 & \text{if } \neg Sm(x) \vee Ca(x) = F \end{cases} \quad (4.181)$$

where the value of $w > 0$ controls strongly we want to enforce the corresponding rule.

The overall joint distribution has the form

$$p(\mathbf{x}) = \frac{1}{Z(\mathbf{w})} \exp\left(\sum_i w_i n_i(\mathbf{x})\right) \quad (4.182)$$

where $n_i(\mathbf{x})$ is the number of instances of clause i which evaluate to true in assignment \mathbf{x} .

Given a grounded MLN model, we can then perform inference using standard methods. Of course, the ground models are often extremely large, so more efficient inference methods, which avoid creating the full ground model (known as **lifted inference**), must be used. See [DL09; KNP11] for details.

One way to gain tractability is to relax the discrete problem to a continuous one. This is the basic idea behind **hinge-loss MRFs** [Bac+15b], which support exact inference using scalable convex optimization. There is a template language for this model family known as **probabilistic soft logic**, which has a similar “flavor” to MLN, although it is not quite as expressive.

Recently MLNs have been combined with DL in various ways. For example, [Zha+20f] uses graph neural networks for inference. And [WP18] uses MLNs for evidence fusion, where the noisy predictions come from DNNs trained using weak supervision.

Finally, it is worth noting one subtlety which arises with undirected models, namely that the size of the unrolled model, which depends on the number of objects in the universe, can affect the results of inference, even if we have no data about the new objects. For example, consider an undirected chain of length T , with T hidden nodes z_t and T observed nodes y_t ; call this model M_1 . Now suppose we double the length of the chain to $2T$, without adding more evidence; call this model M_2 . We find that $p(z_t|y_{1:T}, M_1) \neq p(z_t|y_{1:T}, M_2)$, for $t = 1 : T$, even though we have not added new information, due to the different partition functions. This does not happen with a directed chain, because the newly added nodes can be marginalized out without affecting the original nodes, since the model is locally normalized and therefore modular. See [JBB09; Poo+12] for further discussion.

4.6.5 Open-universe probability models

In Section 4.6.3, we discussed relational probability models, as well as the topic of identity uncertainty. However, we also implicitly made a **closed world assumption**, namely that the set of all objects is fixed and specified ahead of time. In many real world problems, this is an unrealistic assumption. For example, in Section 29.9.3.5, we discuss the problem of tracking an unknown number of objects over time. As another example, consider the problem of enforcing the UN Comprehensive Nuclear Test Ban Treaty (CTBT). This requires monitoring seismic events, and determining if they were caused by nature or man-made explosions. Thus the number of objects of each type, as well as their source, is uncertain [ARS13],

As another (more peaceful) example, suppose we want to perform **citation matching**, in which we want to know whether to cite an arxiv version of a paper or the version on some conference website. Are these the same object? It is often hard to tell, since the titles and author might be the same, yet the content may have been updated. It is often necessary to use subtle cues, such as the date stored in the meta-data, to infer if the two “textual measurements” refer to the same underlying object (paper) or not [Pas+02].

In problems such as these, the number of objects of each type, as well as their relationships, is uncertain. This requires the use of **open-universe probability models** or **OUPM**, which can generate new objects as well as their properties [Rus15; MR10; LB19]. The first formal language for OUPMs was **BLOG** [Mil+05], which stands for “Bayesian LOGic”. This used a general purpose, but slow, MCMC inference scheme to sample over possible worlds of variable size and shape. [Las08; LLC20] describes another open-universe modeling language called **multi-entity Bayesian networks**.

Very recently, Facebook has released the **Bean Machine** library, available at <https://beanmachine.org/>, which supports more efficient inference in OUPMs. Details can be found in [Teh+20], as well as their blog post.¹⁵

4.6.6 Programs as probability models

OUPMs, discussed in Section 4.6.5, let us define probability models over complex dynamic state spaces of unbounded and variable size. The set of possible worlds correspond to objects and their

15. See <https://tinyurl.com/2svy5tmh>.

attributes and relationships. Another approach is to use a **probabilistic programming language** or **PPL**, in which we define the set of possible words as the set of **execution traces** generated by the program when it is endowed with a random choice mechanism. (This is a **procedural approach** to the problem, whereas OUPMs are a **declarative approach**.)

The difference between a probabilistic programming language and a standard one was described in [Gor+14] as follows: “Probabilistic programs are usual functional or imperative programs with two added constructs: (1) the ability to draw values at random from distributions, and (2) the ability to condition values of variables in a program via observation”. The former is a way to define $p(\mathbf{z}, \mathbf{y})$, and the latter is the same as standard Bayesian conditioning $p(\mathbf{z}|\mathbf{y})$.

Some recent examples of PPLs include **Gen** [CT+19], **Pyro** [Bin+19] and **Turing** [GXG18]. Inference in such models is often based on SMC, which we discuss in Chapter 13. For more details on PPLs, see e.g. [Mee+18].

4.7 Structural causal models

While probabilities encode our beliefs about a static world, causality tells us whether and how probabilities change when the world changes, be it by intervention or by act of imagination. — Judea Pearl [PM18b].

In this section, we discuss how we can use directed graphical model notation to represent **causal models**. We discuss causality in greater detail in Chapter 36, but we introduce some basic ideas and notation here, since it is foundational material that we will need in other parts of the book.

The core idea behind causal models is to create a mechanistic model of the world in which we can reason about the effects of local changes. The canonical example is an electronic circuit: we can predict the effects of any action, such as “knocking out” a particular transistor, or changing the resistance level of a wire, by modifying the circuit locally, and then “re-running” it from the same initial conditions.

We can generalize this idea to create a **structural causal models** or **SCM** [PGJ16], also called **functional causal model** [Sch19]. An SCM is a triple $\mathcal{M} = (\mathcal{U}, \mathcal{V}, \mathcal{F})$, where $\mathcal{U} = \{U_i : i = 1 : N\}$ is a set of unexplained or **exogenous** “noise” variables, which are passed as input to the model, $\mathcal{V} = \{V_i : i = 1 : N\}$ is a set of **endogeneous** variables that are part of the model itself, and $\mathcal{F} = \{f_i : i = 1 : N\}$ is a set of deterministic functions of the form $V_i = f_i(V_{\text{pa}_i}, U_i)$, where pa_i are the parents of variable i , and $U_i \in \mathcal{U}$ are the external inputs. We assume the equations can be structured in a **recursive** way, so the dependency graph of nodes given their parents is a DAG. Finally, we assume our model is **causally sufficient**, which means that \mathcal{V} and \mathcal{U} are all of the causally relevant factors (although they may not all be observed). This is called the “**causal Markov assumption**”.

Of course, a model typically cannot represent all the variables that might influence observations or decisions. After all, models are *abstractions* of reality. The variables that we choose not to model explicitly in a functional way can be lumped into the unmodeled exogenous terms. To represent our ignorance about these terms, we can use a distribution $p(U)$ over their values. By “pushing” this external noise through the deterministic part of the model, we induce a distribution over the endogeneous variables, $p(\mathcal{V})$, as in a probabilistic graphical model. However, SCMs make stronger assumptions than PGMs.

We usually assume $p(U)$ is factorized (i.e., the U_i are independent); this is called a **Markovian SCM**. If the exogeneous noise terms are not independent, it would break the assumption that

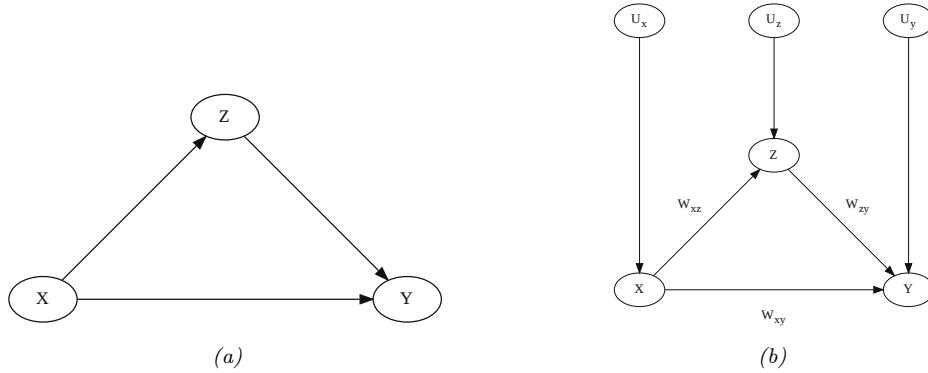


Figure 4.53: (a) PGM for modeling relationship between salary, education and debt. (b) Corresponding SCM.

outcomes can be determined locally using deterministic functions. If there are believed to be dependencies between some of the U_i , we can add extra hidden parents to represent this; this is often depicted as a bidirected or undirected edge connecting the U_i , and is known as a **semi-Markovian SCM**.

4.7.1 Example: causal impact of education on wealth

We now give a simple example of an SCM, based on [PM18b, p276]. Suppose we are interested in the causal effect of education on wealth. Let X represent the level of education of a person (on some numeric scale, say 0 = high school, 1 = college, 2 = graduate school), and Y represent their wealth (at some moment in time). In some cases we might expect that increasing X would increase Y (although it of course depends on the nature of the degree, the nature of the job, etc). Thus we add an edge from X to Y . However, getting more education can cost a lot of money (in certain countries), which is a potentially confounding factor on wealth. Let Z be the debt incurred by a person based on their education. We add an edge from X to Z to reflect the fact that larger X means larger Z (in general), and we add an edge from Z to Y to reflect that larger Z means lower Y (in general).

We can represent our structural assumptions graphically as shown in Figure 4.53b(a). The corresponding SCM has the form:

$$X = f_X(U_x) \tag{4.183}$$

$$Z = f_Z(X, U_z) \tag{4.184}$$

$$Y = f_Y(X, Z, U_y) \tag{4.185}$$

for some set of functions f_x, f_y, f_z , and some prior distribution $p(U_x, U_y, U_z)$. We can also explicitly represent the exogeneous noise terms as shown in Figure 4.53b(b); this makes clear our assumption that the noise terms are a priori independent. (We return to this point later.)

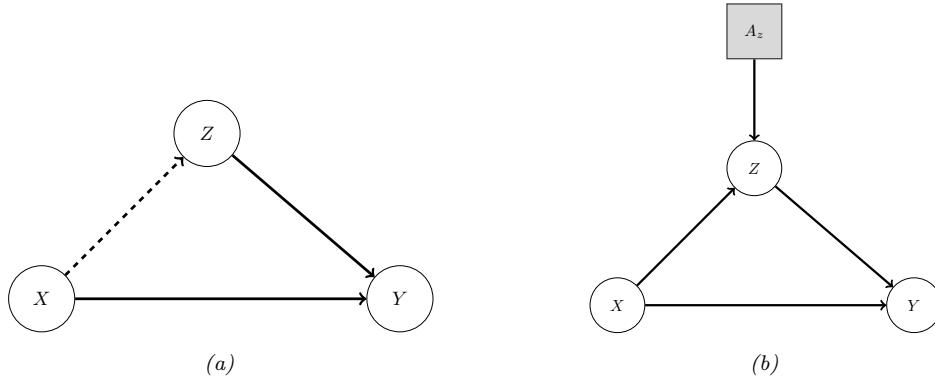


Figure 4.54: An SCM in which we intervene on Z . (a) Hard intervention, in which we clamp Z and thus cut its incoming edges (shown as dotted). (b) Soft intervention, in which we change Z 's mechanism. The square node is an “action” node, using the influence diagram notation from Section 34.2.

4.7.2 Structural equation models

A **structural equation model** [Bol89; BP13], also known as a **path diagram**, is a special case of a structural causal model in which all the functional relationships are linear, and the prior on the noise terms is Gaussian. SEMs are widely used in economics and social science, due to the fact that they have a causal interpretation, yet they are computationally tractable.

For example, let us make an SEM version of our education example. We have

$$X = U_x \tag{4.186}$$

$$Z = c_z + w_{xz}X + U_z \tag{4.187}$$

$$Y = c_y + w_{xy}X + w_{zy}Z + U_y \tag{4.188}$$

If we assume $p(U_x) = \mathcal{N}(U_x|0, \sigma_x^2)$, $p(U_z) = \mathcal{N}(U_z|0, \sigma_z^2)$, and $p(U_y) = \mathcal{N}(U_y|0, \sigma_y^2)$, then the model can be converted to the following Gaussian DGM:

$$p(X) = \mathcal{N}(X|\mu_x, \sigma_x^2) \tag{4.189}$$

$$p(Z|X) = \mathcal{N}(Z|c_z + w_{xz}X, \sigma_z^2) \tag{4.190}$$

$$p(Y|X, Z) = \mathcal{N}(Y|c_y + w_{xy}X + w_{zy}Z, \sigma_y^2) \tag{4.191}$$

We can relax the linearity assumption, to allow arbitrarily flexible functions, and relax the Gaussian assumption, to allow any noise distribution. The resulting “nonparametric SEMs” are equivalent to structural causal models. (For a more detailed comparison between SEMs and SCMs, see [Pea12; BP13; Shi00b].)

4.7.3 Do operator and augmented DAGs

One of the main advantages of SCMs is that they let us predict the effect of **interventions**, which are actions that change one or more local mechanisms. A simple intervention is to force a variable to

have a given value, e.g., we can force a gene to be “on” or “off”. This is called a **perfect intervention** and is written as $\text{do}(X_i = x_i)$, where we have introduced new notation for the “**do**” operator (as in the verb “to do”). This notation means we actively clamp variable X_i to value x_i (as opposed to just observing that it has this value). Since the value of X_i is now independent of its usual parents, we should “cut” the incoming edges to node X_i in the graph. This is called the “**graph surgery**” operation.

In Figure 4.54a we illustrate this for our education SCM, where we force Z to have a given value. For example, we may set $Z = 0$, by paying off everyone’s student debt. Note that $p(X|\text{do}(Z = z)) \neq p(X|Z = z)$, since the intervention changes the model. For example, if we see someone with a debt of 0, we may infer that they probably did not get higher education, i.e., $p(X \geq 1|Z = 0)$ is small; but if we pay off everyone’s college loans, then observing someone with no debt in this modified world should not change our beliefs about whether they got higher education, i.e., $p(X \geq 1|\text{do}(Z = 0)) = p(X \geq 1)$.

In more realistic scenarios, we may not be able to set a variable to a specific value, but we may be able to change it from its current value in some way. For example, we may be able to reduce everyone’s debt by some fixed amount, say $\Delta = -10,000$. Thus we replace $Z = f_Z(X, U_z)$ with $Z = f'_z(Z, U_z)$, where $f'_z(Z, U_z) = f_z(Z, U_z) + \Delta$. This is called an **additive intervention**.

To model this kind of scenario, we can add create an **augmented DAG**, in which every variable is augmented with an additional parent node, representing whether or not the variable’s mechanism is changed in some way [Daw02; Daw15; CPD17]. These extra variables are represented by square nodes, and correspond to decision variables or actions, as in the influence diagram formalism (Section 34.2). The same formalism is used in MDPs for reinforcement learning (see Section 34.5).

We give an example of this in Figure 4.54b, where we add the $A_z \in \{0, 1\}$ node to specify whether we use the debt reduction policy or not. The modified mechanism for Z becomes

$$Z = f'_Z(X, U_x, A_z) = \begin{cases} f_Z(X, U_x) & \text{if } A_z = 0 \\ f_Z(X, U_x) + \Delta & \text{if } A_z = 1 \end{cases} \quad (4.192)$$

With this new definition, conditioning on the effects of an action can be performed using standard probabilistic inference. That is, $p(Q|\text{do}(A_z = a), E = e) = p(Q|A_z = a, E = e)$, where Q is the query (e.g., the event $X \geq 1$) and E are the (possibly empty) evidence variables. This is because the A_z node has no parents, so it has no incoming edges to cut when we clamp it.

Although the augmented DAG allows us to use standard notation (no explicit do operators) and inference machinery, the use of “surgical” interventions, which delete incoming edges to a node that is set to a value, results in a simpler graph, which can simplify many calculations, particularly in the non-parametric setting (see [Pea09b, p361] for a discussion). It is therefore a useful abstraction, even if it is less general than the augmented DAG approach.

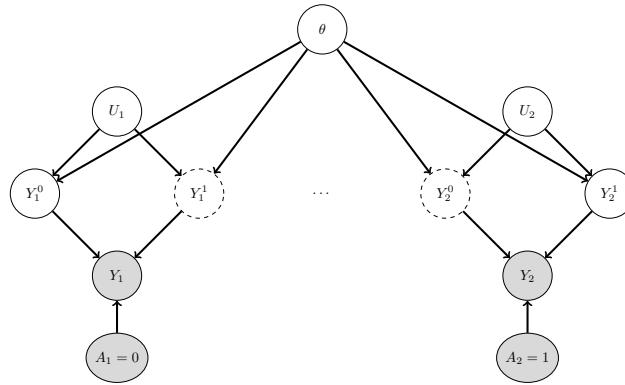
4.7.4 Counterfactuals

So far we have been focused on predicting the **effects of causes**, so we can choose the optimal action (e.g., if I have a headache, I have to decide should I take an aspirin or not). This can be tackled using standard techniques from Bayesian decision theory, as we have seen (see [Daw00; Daw15; LR19; Roh21; DM22] for more details).

Now suppose we are interested in the **causes of effects**. For example, suppose I took the aspirin and my headache did go away. I might be interested in the **counterfactual question** “if I had not

Level	Activity	Questions	Examples
1:Association. $p(Y a)$	Seeing	How would seeing A change my belief in Y ?	Someone took aspirin, how likely is it their headache will be cured?
2:Intervention. $p(Y \text{do}(a))$	Doing	What if I do A ?	If I take aspirin, will my headache be cured?
3:Counterfactuals. $p(Y^a \text{do}(a'), y')$	Imagining	Was it A that caused Y ?	Would my headache be cured had I not taken aspirin?

Table 4.6: Pearl's causal hierarchy. Adapted from Table 1 of [Pea19].

Figure 4.55: Illustration of the potential outcomes framework as a SCM. The nodes with dashed edges are unobserved. In this example, for unit 1, we select action $A_1 = 0$ and observe $Y_1 = Y_1^0 = y_1$, whereas for unit 2, we select action $A_2 = 1$ and observe $Y_2 = Y_2^1 = y_2$.

taken the aspirin, would my headache have gone away anyway?”. This kind of reasoning is crucial for legal reasoning (see e.g., [DMM17]), as well as for tasks like explainability and fairness.

Counterfactual reasoning requires strictly more assumptions than reasoning about interventions (see e.g., [DM22]). Indeed, Judea Pearl has proposed what he calls the **causal hierarchy** [Pea09b; PGJ16; PM18b], which has three levels of analysis, each more powerful than the last, but each making stronger assumptions. See Table 4.6 for a summary.

In counterfactual reasoning, we want to answer questions of the type $p(Y^{a'}|\text{do}(a), y)$, which is read as: “what is the probability distribution over outcomes Y if I were to do a' , given that I have already done a and observed outcome y ”. (We can also condition on any other evidencee that was observed, such as covariates \mathbf{x} .) The quantity $Y^{a'}$ is often called a **potential outcome** [Rub74], since it is the outcome that would occur in a hypothetical world in which you did a' instead of a . (Note that $p(Y^{a'} = y)$ is equivalent to $p(Y = y|\text{do}(a'))$, and is an interventional prediction, not a counterfactual one.)

The assumptions behind the potential outcomes framework can be clearly expressed using a structural causal model. We illustrate this in Figure 4.55 for a simple case where there are two possible actions. We see that we have a set of “units”, such as individual patients, indexed by subscripts. Each unit is associated with a hidden exogeneous random noise source, U_i , that captures

everything that is unique about that unit. This noise gets deterministically mapped to two potential outcomes, Y_i^0 and Y_i^1 , depending on which action is taken. For any given unit, we only get to observe one of the outcomes, namely the one corresponding to the action that was actually chosen. In Figure 4.55, unit 1 chooses action $A_1 = 0$, so we get to see $Y_1^0 = y_1$, whereas unit 2 chooses action $A_2 = 1$, so we get to see $Y_2^1 = y_2$. The fact that we cannot simultaneously see both outcomes for the same unit is called the “**fundamental problem of causal inference**” [Hol86].

We will assume the noise sources are independent, which is known as the “stable unit treatment value assumption” or **SUTVA**. (This would not be true if the treatment on person j could somehow affect the outcome of person i , e.g., due to spreading disease or information between i and j .) We also assume that the deterministic mechanisms that map noise to outcomes are the same across all units (represented by the shared parameter vector θ in Figure 4.55). We need to make one final assumption, namely that the exogeneous noise is not affected by our actions. (This is a formalization of the assumption known as “all else being equal”, or (in legal terms) “**ceteris paribus**”.)

With the above assumptions, we can predict what the outcome *for an individual unit* would have been in the alternative universe where we picked the other action. The procedure is as follows. First we perform **abduction** using SCM G , to infer $p(U_i|A_i = a, Y_i = y_i)$, which is the posterior over the latent factors for unit i given the observed evidence in the actual world. Second we perform **intervention**, in which we modify the causal mechanisms of G by replacing $A_i = a$ with $A_i = a'$ to get $G_{a'}$. Third we perform **prediction**, in which we propagate the distribution of the latent factors, $p(U_i|A_i = a, Y_i = y_i)$, through the modified SCM $G_{a'}$ to get $p(Y_i^{a'}|A_i = a, Y_i = y_i)$.

In Figure 4.55, we see that we have two copies of every possible outcome variable, to represent the set of possible worlds. Of course, we only get to see one such world, based on the actions that we actually took. More generally, a model in which we “clone” all the deterministic variables, with the noise being held constant between the two branches of the graph for the same unit, is called a **twin network** [Pea09b]. We will see a more practical example in Section 29.12.6, where we discuss assessing the counterfactual causal impact of an intervention in a time series. (See also [RR11; RR13], who propose a related formalism known as **single world intervention graph** or **SWIG**.)

We see from the above that the potential outcomes framework is mathematically equivalent to structural causal models, but does not use graphical model notation. This has led to heated debate between the founders of the two schools of thought.¹⁶ The SCM approach is more popular in computer science (see e.g., [PJS17; Sch19; Sch+21b]), and the PO approach is more popular in economics (see e.g., [AP09; Imb19]). Modern textbooks on causality usually use both formalisms (see e.g., [HR20a; Nea20]).

¹⁶. The potential outcomes framework is based on the work of Donald Rubin, and others, and is therefore sometimes called the **Rubin causal model** (see e.g., https://en.wikipedia.org/wiki/Rubin_causal_model). The structural causal models framework is based on the work of Judea Pearl and others. See e.g., <http://causality.cs.ucla.edu/blog/index.php/2012/12/03/judea-pearl-on-potential-outcomes/> for a discussion of the two.

5 Information theory

Machine learning is fundamentally about **information processing**. But what is information anyway, and how do we measure it? Ultimately we need a way to quantify the magnitude of an update from one set of beliefs to another. It turns out that with a relatively short list of desiderata there is a unique answer: the Kullback-Leibler (KL) divergence (see Section 5.1). We'll study the properties of the KL divergence and two special cases: entropy (Section 5.2), and mutual information (Section 5.3). that are useful enough to merit independent study. We then go on to briefly discuss two main applications of information theory. The first application is **data compression** or **source coding**, which is the problem of removing redundancy from data so it can be represented more compactly, either in a lossless way (e.g., ZIP files) or a lossy way (e.g., MP3 files). See Section 5.4 for details. The second application is **error correction** or **channel coding**, which means encoding data in such a way that it is robust to errors when sent over a noisy channel, such as a telephone line or a satellite link. See Section 5.5 for details.

It turns out that methods for data compression and error correction both rely on having an accurate probabilistic model of the data. For compression, a probabilistic model is needed so the sender can assign shorter **codewords** to data vectors which occur most often, and hence save space. For error correction, a probabilistic model is needed so the receiver can infer the most likely source message by combining the received noisy message with a prior over possible messages.

It is clear that probabilistic machine learning is useful for information theory. However, information theory is also useful for machine learning. Indeed, we have seen that Bayesian machine learning is about representing and reducing our uncertainty, and so is fundamentally about information. In Section 5.6.2, we explore this direction in more detail, where we discuss the information bottleneck.

For more information on information theory, see e.g., [Mac03; CT06].

5.1 KL divergence

This section is written with Alex Alemi.

To discuss information theory, we need some way to measure or quantify information itself. Let's say we start with some distribution describing our degrees of belief about a random variable, call it $q(x)$. We then want to update our degrees of belief to some new distribution $p(x)$, perhaps because we've taken some new measurements or merely thought about the problem a bit longer. What we seek is a mathematical way to quantify the magnitude of this update, which we'll denote $I[p||q]$. What sort of criteria would be reasonable for such a measure? We discuss this issue below, and then define a quantity that satisfies these criteria.

5.1.1 Desiderata

For simplicity, imagine we are describing a distribution over N possible events. In this case, the probability distribution $q(\mathbf{x})$ consists of N non-negative real numbers that add up to 1. To be even more concrete, imagine we are describing the random variable representing the suit of the next card we'll draw from a deck: $S \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$. Imagine we initially believe the distributions over suits to be uniform: $q = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$. If our friend told us they removed all of the red cards we could update to: $q' = [\frac{1}{2}, \frac{1}{2}, 0, 0]$. Alternatively, we might believe some diamonds changed into clubs and want to update to $q'' = [\frac{3}{8}, \frac{2}{8}, \frac{2}{8}, \frac{1}{8}]$. Is there a good way to quantify *how much* we've updated our beliefs? Which is a larger update: $q \rightarrow q'$ or $q \rightarrow q''$?

It seems desireable that any useful such measure would satisfy the following properties:

1. *continuous* in its arguments: If we slightly perturb either our starting or ending distribution, it should similarly have a small effect on the magnitude of the update. For example: $I[p \parallel \frac{1}{4} + \epsilon, \frac{1}{4}, \frac{1}{4}, \frac{1}{4} - \epsilon]$ should be close to $I[p \parallel q]$ for small ϵ , where $q = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$.
2. *non-negative*: $I[p \parallel q] \geq 0$ for all $p(\mathbf{x})$ and $q(\mathbf{x})$. The magnitude of our updates are non-negative.
3. *permutation invariant*: The magnitude of the update should not depend on the order we choose for the elements of \mathbf{x} . For example, it shouldn't matter if I list my probabilities for the suits of cards in the order $\clubsuit, \spadesuit, \heartsuit, \diamondsuit$ or $\clubsuit, \diamondsuit, \heartsuit, \spadesuit$, if I keep the order consistent across all of the distributions, I should get the same answer. For example: $I[a, b, c, d \parallel e, f, g, h] = I[a, d, c, b \parallel e, h, g, f]$.
4. *monotonic* for uniform distributions: While it's hard to say how large the updates in our beliefs are in general, there are some special cases for which we have a strong intuition. If our beliefs update from a uniform distribution on N elements to one that is uniform in N' elements, the information gain should be an increasing function of N and a decreasing function of N' . For instance changing from a uniform distribution on all four suits $[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$ (so $N = 4$) to only one suit, such as all clubs, $[1, 0, 0, 0]$ where $N' = 1$, is a larger update than if I only updated to the card being black, $[\frac{1}{2}, \frac{1}{2}, 0, 0]$ where $N' = 2$.
5. satisfy a natural *chain rule*: So far we've been describing our beliefs in what will happen on the next card draw as a single random variable representing the suit of the next card ($S \in \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$). We could equivalently describe the same physical process in two steps. First we consider the random variable representing the color of the card ($C \in \{\blacksquare, \square\}$), which could be either black ($\blacksquare = \{\clubsuit, \spadesuit\}$) or red ($\square = \{\heartsuit, \diamondsuit\}$). Then, if we draw a red card we describe our belief that it is \heartsuit versus \diamondsuit . If it was instead black we would assign beliefs to it being \clubsuit versus \spadesuit . We can convert any distribution over the four suits into this conditional factorization, for example:

$$p(S) = \left[\frac{3}{8}, \frac{2}{8}, \frac{2}{8}, \frac{1}{8} \right] \tag{5.1}$$

becomes

$$p(C) = \left[\frac{5}{8}, \frac{3}{8} \right] \quad p(\{\clubsuit, \spadesuit\} | C = \blacksquare) = \left[\frac{3}{5}, \frac{2}{5} \right] \quad p(\{\heartsuit, \diamondsuit\} | C = \square) = \left[\frac{2}{3}, \frac{1}{3} \right]. \tag{5.2}$$

In the same way we could decompose our uniform distribution q . Obviously, for our measure of information to be of use the magnitude of the update needs to be the same regardless of how we

choose to describe what is ultimately the same physical process. What we need is some way to relate what would be four different invocations of our information function:

$$I_S \equiv I[p(S)\|q(S)] \quad (5.3)$$

$$I_C \equiv I[p(C)\|q(C)] \quad (5.4)$$

$$I_{\blacksquare} \equiv I[p(\{\clubsuit, \spadesuit\}|C = \blacksquare)\|q(\{\clubsuit, \spadesuit\}|C = \blacksquare)] \quad (5.5)$$

$$I_{\square} \equiv I[p(\{\heartsuit, \diamondsuit\}|C = \square)\|q(\{\heartsuit, \diamondsuit\}|C = \square)]. \quad (5.6)$$

Clearly I_S should be some function of $\{I_C, I_{\blacksquare}, I_{\square}\}$. Our last desideratum is that the way we measure the magnitude of our updates will have I_S be a linear combination of $I_C, I_{\blacksquare}, I_{\square}$. In particular, we will require that they combine as a weighted linear combinations, with weights set by the probability that we would find ourselves in that branch according to the distribution p :

$$I_S = I_C + p(C = \blacksquare)I_{\blacksquare} + p(C = \square)I_{\square} = I_C + \frac{5}{8}I_{\blacksquare} + \frac{3}{8}I_{\square} \quad (5.7)$$

Stating this requirement more generally: If we partition \mathbf{x} into two pieces $[\mathbf{x}_L, \mathbf{x}_R]$, so that we can write $p(\mathbf{x}) = p(\mathbf{x}_L)p(\mathbf{x}_R|\mathbf{x}_L)$ and similarly for q , the magnitude of the update should be

$$I[p(\mathbf{x})\|q(\mathbf{x})] = I[p(\mathbf{x}_L)\|q(\mathbf{x}_L)] + \mathbb{E}_{p(\mathbf{x}_L)}[I[p(\mathbf{x}_R|\mathbf{x}_L)\|q(\mathbf{x}_R|\mathbf{x}_L)]] . \quad (5.8)$$

Notice that this requirement *breaks the symmetry between our two distributions*: The right hand side asks us to take the expected conditional information gain with respect to the marginal, but we need to decide which of two marginals to take the expectation with respect to.

5.1.2 The KL divergence uniquely satisfies the desiderata

We will now define a quantity that is the only measure (up to a multiplicative constant) that satisfies the above desiderata. The **Kullback-Leibler divergence** or **KL divergence**, also known as the **information gain** or **relative entropy**, is defined as follows:

$$D_{\text{KL}}(p \parallel q) \triangleq \sum_{k=1}^K p_k \log \frac{p_k}{q_k} . \quad (5.9)$$

This naturally extends to continuous distributions:

$$D_{\text{KL}}(p \parallel q) \triangleq \int dx p(x) \log \frac{p(x)}{q(x)} . \quad (5.10)$$

Next we will verify that this definition satisfies all of our desiderata. (The proof that it is the unique measure which captures these properties can be found in, e.g., [Hob69; Rén61].)

5.1.2.1 Continuity of KL

One of our desiderata was that our measure of information gain should be continuous. The KL divergence is manifestly continuous in its arguments except potentially when p_k or q_k is zero. In the first case, notice that the limit as $p \rightarrow 0$ is well behaved:

$$\lim_{p \rightarrow 0} p \log \frac{p}{q} = 0 . \quad (5.11)$$

Taking this as the definition of the value of the integrand when $p = 0$ will make it continuous there. Notice that we do have a problem however if $q = 0$ in some place that $p \neq 0$. Our information gain requires that our original distribution of beliefs q has some support everywhere the updated distribution does. Intuitively it would require an infinite amount of information for us to update our beliefs in some outcome to change from being exactly 0 to some positive value.

5.1.2.2 Non-negativity of KL divergence

In this section, we prove that the KL divergence as defined is always non-negative. We will make use of **Jensen's inequality**, which states that for any convex function f , we have that

$$f\left(\sum_{i=1}^n \lambda_i \mathbf{x}_i\right) \leq \sum_{i=1}^n \lambda_i f(\mathbf{x}_i) \quad (5.12)$$

where $\lambda_i \geq 0$ and $\sum_{i=1}^n \lambda_i = 1$. This can be proved by induction, where the base case with $n = 2$ follows by definition of convexity.

Theorem 5.1.1. (*Information inequality*) $D_{\text{KL}}(p \parallel q) \geq 0$ with equality iff $p = q$.

Proof. We now prove the theorem, following [CT06, p28]. As we noted in the previous section, the KL divergence requires special consideration when $p(x)$ or $q(x) = 0$, the same is true here. Let $A = \{x : p(x) > 0\}$ be the support of $p(x)$. Using the convexity of the log function and Jensen's inequality, we have that

$$-D_{\text{KL}}(p \parallel q) = -\sum_{x \in A} p(x) \log \frac{p(x)}{q(x)} = \sum_{x \in A} p(x) \log \frac{q(x)}{p(x)} \quad (5.13)$$

$$\leq \log \sum_{x \in A} p(x) \frac{q(x)}{p(x)} = \log \sum_{x \in A} q(x) \quad (5.14)$$

$$\leq \log \sum_{x \in \mathcal{X}} q(x) = \log 1 = 0 \quad (5.15)$$

Since $\log(x)$ is a strictly concave function ($-\log(x)$ is convex), we have equality in Equation (5.14) iff $p(x) = cq(x)$ for some c that tracks the fraction of the whole space \mathcal{X} contained in A . We have equality in Equation (5.15) iff $\sum_{x \in A} q(x) = \sum_{x \in \mathcal{X}} q(x) = 1$, which implies $c = 1$. Hence $D_{\text{KL}}(p \parallel q) = 0$ iff $p(x) = q(x)$ for all x . \square

The non-negativity of KL divergence often feels as though it's one of the most useful results in information theory. It is a good result to keep in your back pocket. Anytime you can rearrange an expression in terms of KL divergence terms, since those are guaranteed to be non-negative, dropping them immediately generates a bound.

5.1.2.3 KL divergence is invariant to reparameterizations

We wanted our measure of information to be invariant to permutations of the labels. The discrete form is manifestly permutation invariant as summations are. The KL divergence actually satisfies a

much stronger property of reparameterization invariance. Namely, we can transform our random variable through an arbitrary invertible map and it won't change the value of the KL divergence.

If we transform our random variable from x to some $y = f(x)$ we know that $p(x) dx = p(y) dy$ and $q(x) dx = q(y) dy$. Hence the KL divergence remains the same for both random variables:

$$D_{\text{KL}}(p(x) \parallel q(x)) = \int dx p(x) \log \frac{p(x)}{q(x)} = \int dy p(y) \log \left(\frac{p(y) \left| \frac{dy}{dx} \right|}{q(y) \left| \frac{dy}{dx} \right|} \right) = D_{\text{KL}}(p(y) \parallel q(y)). \quad (5.16)$$

Because of this reparameterization invariance we can rest assured that when we measure the KL divergence between two distributions we are measuring something about the distributions and not the way we choose to represent the space in which they are defined. We are therefore free to transform our data into a convenient basis of our choosing, such as a Fourier bases for images, without affecting the result.

5.1.2.4 Monotonicity for uniform distributions

Consider updating a probability distribution from a uniform distribution on N elements to a uniform distribution on N' elements. The KL divergence is:

$$D_{\text{KL}}(p \parallel q) = \sum_k \frac{1}{N'} \log \frac{\frac{1}{N'}}{\frac{1}{N}} = \log \frac{N}{N'}, \quad (5.17)$$

or the log of the ratio of the elements before and after the update. This satisfies our monotonicity requirement.

We can interpret this result as follows: Consider finding an element of a sorted array by means of bisection. A well designed yes/no question can cut the search space in half. Measured in bits, the KL divergence tells us how many well designed yes/no questions are required on average to move from q to p .

5.1.2.5 Chain rule for KL divergence

Here we show that the KL divergence satisfies a natural chain rule:

$$D_{\text{KL}}(p(x, y) \parallel q(x, y)) = \int dx dy p(x, y) \log \frac{p(x, y)}{q(x, y)} \quad (5.18)$$

$$= \int dx dy p(x, y) \left[\log \frac{p(x)}{q(x)} + \log \frac{p(y|x)}{q(y|x)} \right] \quad (5.19)$$

$$= D_{\text{KL}}(p(x) \parallel q(x)) + \mathbb{E}_{p(x)} [D_{\text{KL}}(p(y|x) \parallel q(y|x))]. \quad (5.20)$$

We can rest assured that we can decompose our distributions into their conditionals and the KL divergences will just add.

As a notational convenience, the **conditional KL divergence** is defined to be the expected value of the KL divergence between two conditional distributions:

$$D_{\text{KL}}(p(y|x) \parallel q(y|x)) \triangleq \int dx p(x) \int dy p(y|x) \log \frac{p(y|x)}{q(y|x)}. \quad (5.21)$$

This allows us to drop many expectation symbols.

5.1.3 Thinking about KL

In this section, we discuss some qualitative properties of the KL divergence.

5.1.3.1 Units of KL

Above we said that the desiderata we listed determined the KL divergence up to a multiplicative constant. Because the KL divergence is logarithmic, and logarithms in different bases are the same up to a multiplicative constant, our choice of the base of the logarithm when we compute the KL divergence is a choice akin to choosing which units to measure the information in.

If the KL divergence is measured with the base-2 logarithm, it is said to have units of **bits**, short for “binary digits”. If measured using the natural logarithm as we normally do for mathematical convenience, it is said to be measured in **nats** for “natural units”.

To convert between the systems, we use $\log_2 y = \frac{\log y}{\log 2}$. Hence

$$1 \text{ bit} = \log 2 \text{ nats} \sim 0.693 \text{ nats} \quad (5.22)$$

$$1 \text{ nat} = \frac{1}{\log 2} \text{ bits} \sim 1.44 \text{ bits}. \quad (5.23)$$

5.1.3.2 Asymmetry of the KL divergence

The KL divergence is *not* symmetric in its two arguments. While many find this asymmetry confusing at first, we can see that the asymmetry stems from our requirement that we have a natural chain rule. When we decompose the distribution into its conditional, we need to take an expectation with respect to the variables being conditioned on. In the KL divergence we take this expectation with respect to the first argument $p(x)$. This breaks the symmetry between the two distributions.

At a more intuitive level, we can see that the information required to move from q to p is in general different than the information required to move from p to q . For example, consider the KL divergence between two Bernoulli distributions, the first with the probability of success given by 0.443 and the second with 0.975:

$$D_{\text{KL}} = 0.975 \log \frac{0.975}{0.443} + 0.025 \log \frac{0.025}{0.557} = 0.692 \text{ nats} \sim 1.0 \text{ bits}. \quad (5.24)$$

So it takes 1 bit of information to update from a [0.443, 0.557] distribution to a [0.975, 0.025] Bernoulli distribution. What about the reverse?

$$D_{\text{KL}} = 0.443 \log \frac{0.443}{0.975} + 0.557 \log \frac{0.557}{0.025} = 1.38 \text{ nats} \sim 2.0 \text{ bits}, \quad (5.25)$$

so it takes two bits, or twice as much information to move the other way. Thus we see that starting with a distribution that is nearly even and moving to one that is nearly certain takes about 1 bit of information, or one well designed yes/no question. To instead move us from near certainty in an outcome to something that is akin to the flip of a coin requires more persuasion.

5.1.3.3 KL as expected weight of evidence

Imagine you have two different hypotheses you wish to select between, which we'll label P and Q . You collect some data D . Bayes' rule tells us how to update our beliefs in the hypotheses being

correct:

$$\Pr(P|D) = \frac{\Pr(D|P)}{\Pr(D)} \Pr(P). \quad (5.26)$$

Normally this requires being able to evaluate the marginal likelihood $\Pr(D)$, which is difficult. If we instead consider the ratio of the probabilities for the two hypotheses:

$$\frac{\Pr(P|D)}{\Pr(Q|D)} = \frac{\Pr(D|P)}{\Pr(D|Q)} \frac{\Pr(P)}{\Pr(Q)}, \quad (5.27)$$

the marginal likelihood drops out. Taking the logarithm of both sides, and identifying the probability of the data under the model as the likelihood we find:

$$\log \frac{\Pr(P|D)}{\Pr(Q|D)} = \log \frac{p(D)}{q(D)} + \log \frac{\Pr(P)}{\Pr(Q)}. \quad (5.28)$$

The posterior log probability ratio for one hypothesis over the other is just our prior log probability ratio plus a term that I. J. Good called the **weight of evidence** [Goo85] D for hypothesis P over Q :

$$w[P/Q; D] \triangleq \log \frac{p(D)}{q(D)}. \quad (5.29)$$

With this interpretation, the KL divergence is the expected weight of evidence for P over Q given by each observation, provided P were correct. Thus we see that data will (on average) add rather than subtract evidence towards the correct hypothesis, since KL divergence is always non-negative in expectation (see Section 5.1.2.2).

5.1.4 Minimizing KL

In this section, we discuss ways to minimize $D_{\text{KL}}(p \parallel q)$ or $D_{\text{KL}}(q \parallel p)$ wrt an approximate distribution q , given a true distribution p .

5.1.4.1 Forwards vs reverse KL

The asymmetry of KL means that finding a q that is close to p by minimizing $D_{\text{KL}}(p \parallel q)$ (also called the **inclusive KL** or **forwards KL**) gives different behavior than minimizing $D_{\text{KL}}(q \parallel p)$ (also called the **exclusive KL** or **reverse KL**). For example, consider the bimodal distribution p shown in blue in Figure 5.1, which we approximate with a unimodal Gaussian q .

To prevent $D_{\text{KL}}(p \parallel q)$ from becoming infinite, we must have $q > 0$ whenever $p > 0$ (i.e., q must have support everywhere p does), so q tends to *cover* both modes as it must be nonvanishing everywhere p is; this is called **mode-covering** or **zero-avoiding** behavior (orange curve). By contrast, to prevent $D_{\text{KL}}(q \parallel p)$ from becoming infinite, we must have $q = 0$ whenever $p = 0$, which creates **mode-seeking** or **zero-forcing** behavior (green curve).

For an animated visualization (written by Ari Seff) of the difference between these two objectives, see https://twitter.com/ari_seff/status/1303741288911638530.

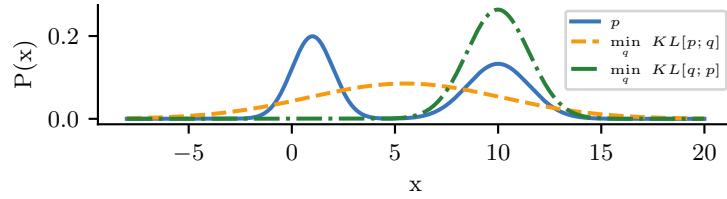


Figure 5.1: Demonstration of the mode-covering or mode-seeking behavior of KL divergence. The original distribution p (shown in blue) is bimodal. When we minimize $D_{\text{KL}}(p \parallel q)$, then q covers the modes of p (orange). When we minimize $D_{\text{KL}}(q \parallel p)$, then q ignores some of the modes of p (green). Generated by `minimize_kl_divergence.ipynb`.

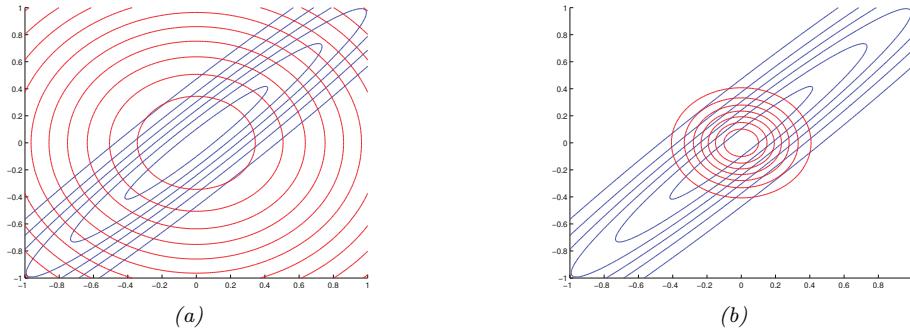


Figure 5.2: Illustrating forwards vs reverse KL on a symmetric Gaussian. The blue curves are the contours of the true distribution p . The red curves are the contours of a factorized approximation q . (a) Minimizing $D_{\text{KL}}(p \parallel q)$. (b) Minimizing $D_{\text{KL}}(q \parallel p)$. Adapted from Figure 10.2 of [Bis06]. Generated by `kl_pq_gauss.ipynb`.

5.1.4.2 Moment projection (mode covering)

Suppose we compute q by minimizing the forwards KL:

$$q = \underset{q}{\operatorname{argmin}} D_{\text{KL}}(p \parallel q) \quad (5.30)$$

This is called **M-projection**, or **moment projection** since the optimal q matches the moments of p , as we show below. The process of computing q is therefore called **moment matching**.

To see why the optimal q must match the moments of p , let us assume that q is an exponential family distribution of the form

$$q(\mathbf{x}) = h(\mathbf{x}) \exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})] \quad (5.31)$$

where $\mathcal{T}(\mathbf{x})$ is the vector of sufficient statistics, and $\boldsymbol{\eta}$ are the natural parameters. The first order

optimality conditions are as follows:

$$\partial_{\eta_i} D_{\text{KL}}(p \parallel q) = -\partial_{\eta_i} \int_{\mathbf{x}} p(\mathbf{x}) \log q(\mathbf{x}) \quad (5.32)$$

$$= -\partial_{\eta_i} \int_{\mathbf{x}} p(\mathbf{x}) \log (h(\mathbf{x}) \exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})]) \quad (5.33)$$

$$= -\partial_{\eta_i} \int_{\mathbf{x}} p(\mathbf{x}) (\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - \log Z(\boldsymbol{\eta})) \quad (5.34)$$

$$= - \int_{\mathbf{x}} p(\mathbf{x}) \mathcal{T}_i(\mathbf{x}) + \mathbb{E}_{q(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] \quad (5.35)$$

$$= -\mathbb{E}_{p(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] + \mathbb{E}_{q(\mathbf{x})} [\mathcal{T}_i(\mathbf{x})] = 0 \quad (5.36)$$

where in the penultimate line we used the fact that the derivative of the log partition function yields the expected sufficient statistics, as shown in Equation (2.216). Hence the expected sufficient statistics (moments of the distribution) must match.

As an example, suppose the true target distribution p is a correlated 2d Gaussian, $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1})$, where

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix} \quad \boldsymbol{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{pmatrix} \quad (5.37)$$

We will approximate this with a distribution q which is a product of two 1d Gaussians, i.e., a Gaussian with a diagonal covariance matrix:

$$q(\mathbf{x}|\mathbf{m}, \mathbf{V}) = \mathcal{N}(x_1|m_1, v_1) \mathcal{N}(x_2|m_2, v_2) \quad (5.38)$$

If we perform moment matching, the optimal q must therefore have the following form:

$$q(\mathbf{x}) = \mathcal{N}(x_1|\mu_1, \Sigma_{11}) \mathcal{N}(x_2|\mu_2, \Sigma_{22}) \quad (5.39)$$

In Figure 5.2(a), we show the resulting distribution. We see that q covers (includes) p , but its support is too broad (under-confidence).

5.1.4.3 Information projection (mode seeking)

Now suppose we compute q by minimizing the reverse KL:

$$q = \underset{q}{\operatorname{argmin}} D_{\text{KL}}(q \parallel p) \quad (5.40)$$

This is called **I-projection**, or **information projection**. This optimization problem is often easier to compute, since the objective requires taking expectations wrt q , which we can choose to be a tractable family.

As an example, consider again the case where the true distribution is a full covariance Gaussian, $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1})$, and let the approximation be a diagonal Gaussian, $q(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{m}, \text{diag}(\mathbf{v}))$. Then one can show (see [Supplementary](#) Section 5.1.2) that the optimal variational parameters are $\mathbf{m} = \boldsymbol{\mu}$ and $v_i = \Lambda_{ii}^{-1}$. We illustrate this in 2d in Figure 5.2(b). We see that the posterior variance is too narrow, i.e, the approximate posterior is overconfident. Note, however, that minimizing the reverse KL does not always result in an overly compact approximation, as explained in [Tur+08].

5.1.5 Properties of KL

Below are some other useful properties of the KL divergence.

5.1.5.1 Compression lemma

An important general purpose result for the KL divergence is the **compression lemma**:

Theorem 5.1.2. *For any distributions P and Q with a well-defined KL divergence, and for any scalar function ϕ defined on the domain of the distributions we have that:*

$$\mathbb{E}_P [\phi] \leq \log \mathbb{E}_Q [e^\phi] + D_{\text{KL}}(P \| Q). \quad (5.41)$$

Proof. We know that the KL divergence between any two distributions is non-negative. Consider a distribution of the form:

$$g(x) = \frac{q(x)}{\mathcal{Z}} e^{\phi(x)}. \quad (5.42)$$

where the *partition function* is given by:

$$\mathcal{Z} = \int dx q(x) e^{\phi(x)}. \quad (5.43)$$

Taking the KL divergence between $p(x)$ and $g(x)$ and rearranging gives the bound:

$$D_{\text{KL}}(P \| G) = D_{\text{KL}}(P \| Q) - \mathbb{E}_P [\phi(x)] + \log(\mathcal{Z}) \geq 0. \quad (5.44)$$

□

One way to view the compression lemma is that it provides what is termed the Donsker-Varadhan variational representation of the KL divergence:

$$D_{\text{KL}}(P \| Q) = \sup_{\phi} \mathbb{E}_P [\phi(x)] - \log \mathbb{E}_Q [e^{\phi(x)}]. \quad (5.45)$$

In the space of all possible functions ϕ defined on the same domain as the distributions, assuming all of the values above are finite, the KL divergence is the supremum achieved. For any fixed function $\phi(x)$, the right hand side provides a lower bound on the true KL divergence.

Another use of the compression lemma is that it provides a way to estimate the expectation of some function with respect to an unknown distribution P . In this spirit, the compression lemma can be used to power a set of what are known as PAC-Bayes bounds of losses with respect to the true distribution in terms of measured losses with respect to a finite training set. See for example Section 17.4.5 or Banerjee [Ban06].

5.1.5.2 Data processing inequality for KL

We now show that any processing we do on samples from two different distributions makes their samples approach one another. This is called the **data processing inequality**, since it shows that we cannot increase the information gain from q to p by processing our data and then measuring it.

Theorem 5.1.3. *Consider two different distributions $p(x)$ and $q(x)$ combined with a probabilistic channel $t(y|x)$. If $p(y)$ is the distribution that results from sending samples from $p(x)$ through the channel $t(y|x)$ and similarly for $q(y)$ we have that:*

$$D_{\text{KL}}(p(x) \parallel q(x)) \geq D_{\text{KL}}(p(y) \parallel q(y)) \quad (5.46)$$

Proof. The proof uses Jensen's inequality from Section 5.1.2.2 again. Call $p(x, y) = p(x)t(y|x)$ and $q(x, y) = q(x)t(y|x)$.

$$D_{\text{KL}}(p(x) \parallel q(x)) = \int dx p(x) \log \frac{p(x)}{q(x)} \quad (5.47)$$

$$= \int dx \int dy p(x)t(y|x) \log \frac{p(x)t(y|x)}{q(x)t(y|x)} \quad (5.48)$$

$$= \int dx \int dy p(x, y) \log \frac{p(x, y)}{q(x, y)} \quad (5.49)$$

$$= - \int dy p(y) \int dx p(x|y) \log \frac{q(x, y)}{p(x, y)} \quad (5.50)$$

$$\geq - \int dy p(y) \log \left(\int dx p(x|y) \frac{q(x, y)}{p(x, y)} \right) \quad (5.51)$$

$$= - \int dy p(y) \log \left(\frac{q(y)}{p(y)} \int dx q(x|y) \right) \quad (5.52)$$

$$= \int dy p(y) \log \frac{p(y)}{q(y)} = D_{\text{KL}}(p(y) \parallel q(y)) \quad (5.53)$$

□

One way to interpret this result is that any processing done to random samples makes it harder to tell two distributions apart.

As a special form of processing, we can simply marginalize out a subset of random variables.

Corollary 5.1.1. (*Monotonicity of KL divergence*)

$$D_{\text{KL}}(p(x, y) \parallel q(x, y)) \geq D_{\text{KL}}(p(x) \parallel q(x)) \quad (5.54)$$

Proof. The proof is essentially the same as the one above.

$$D_{\text{KL}}(p(x, y) \parallel q(x, y)) = \int dx \int dy p(x, y) \log \frac{p(x, y)}{q(x, y)} \quad (5.55)$$

$$= - \int dy p(y) \int dx p(x|y) \log \left(\frac{q(y)}{p(y)} \frac{q(x|y)}{p(x|y)} \right) \quad (5.56)$$

$$\geq - \int dy p(y) \log \left(\frac{q(y)}{p(y)} \int dx q(x|y) \right) \quad (5.57)$$

$$= \int dy p(y) \log \frac{p(y)}{q(y)} = D_{\text{KL}}(p(y) \parallel q(y)) \quad (5.58)$$

(5.59)

□

One intuitive interpretation of this result is that if you only partially observe random variables, it is harder to distinguish between two candidate distributions than if you observed all of them.

5.1.6 KL divergence and MLE

Suppose we want to find the distribution q that is as close as possible to p , as measured by KL divergence:

$$q^* = \arg \min_q D_{\text{KL}}(p \parallel q) = \arg \min_q \int p(x) \log p(x) dx - \int p(x) \log q(x) dx \quad (5.60)$$

Now suppose p is the empirical distribution, which puts a probability atom on the observed training data and zero mass everywhere else:

$$p_{\mathcal{D}}(x) = \frac{1}{N} \sum_{n=1}^N \delta(x - x_n) \quad (5.61)$$

Using the sifting property of delta functions we get

$$D_{\text{KL}}(p_{\mathcal{D}} \parallel q) = - \int p_{\mathcal{D}}(x) \log q(x) dx + C \quad (5.62)$$

$$= - \int \left[\frac{1}{N} \sum_n \delta(x - x_n) \right] \log q(x) dx + C \quad (5.63)$$

$$= - \frac{1}{N} \sum_n \log q(x_n) + C \quad (5.64)$$

where $C = \int p_{\mathcal{D}}(x) \log p_{\mathcal{D}}(x)$ is a constant independent of q .

We can rewrite the above as follows

$$D_{\text{KL}}(p_{\mathcal{D}} \parallel q) = \mathbb{H}_{ce}(p_{\mathcal{D}}, q) - \mathbb{H}(p_{\mathcal{D}}) \quad (5.65)$$

where

$$\mathbb{H}_{ce}(p, q) \triangleq -\sum_k p_k \log q_k \quad (5.66)$$

is known as the **cross entropy**. The quantity $\mathbb{H}_{ce}(p_D, q)$ is the average negative log likelihood of q evaluated on the training set. Thus we see that minimizing KL divergence to the empirical distribution is equivalent to maximizing likelihood.

This perspective points out the flaw with likelihood-based training, namely that it puts too much weight on the training set. In most applications, we do not really believe that the empirical distribution is a good representation of the true distribution, since it just puts “spikes” on a finite set of points, and zero density everywhere else. Even if the dataset is large (say 1M images), the universe from which the data is sampled is usually even larger (e.g., the set of “all natural images” is much larger than 1M). Thus we need to somehow smooth the empirical distribution by sharing probability mass between “similar” inputs.

5.1.7 KL divergence and Bayesian inference

Bayesian inference itself can be motivated as the solution to a particular minimization problem of KL.

Consider a prior set of beliefs described by a joint distribution $q(\theta, D) = q(\theta)q(D|\theta)$, involving some *prior* $q(\theta)$ and some *likelihood* $q(D|\theta)$. If we happen to observe some particular dataset D_0 , how should we update our beliefs? We could search for the joint distribution that is as close as possible to our prior beliefs but that respects the constraint that we now know the value of the data:

$$p(\theta, D) = \operatorname{argmin} D_{\text{KL}}(p(\theta, D) \parallel q(\theta, D)) \text{ such that } p(D) = \delta(D - D_0). \quad (5.67)$$

where $\delta(D - D_0)$ is a degenerate distribution that puts all its mass on the dataset D that is identically equal to D_0 . Writing the KL out in its chain rule form:

$$D_{\text{KL}}(p(\theta, D) \parallel q(\theta, D)) = D_{\text{KL}}(p(D) \parallel q(D)) + D_{\text{KL}}(p(\theta|D) \parallel q(\theta|D)), \quad (5.68)$$

makes clear that the solution is given by the joint distribution:

$$p(\theta, D) = p(D)p(\theta|D) = \delta(D - D_0)q(\theta|D). \quad (5.69)$$

Our updated beliefs have a marginal over the θ

$$p(\theta) = \int dD p(\theta, D) = \int dD \delta(D - D_0)q(\theta|D) = q(\theta|D = D_0), \quad (5.70)$$

which is just the usual Bayesian posterior from our prior beliefs evaluated at the data we observed.

By contrast, the usual statement of Bayes’ rule is just a trivial observation about the chain rule of probabilities:

$$q(\theta, D) = q(D)q(\theta|D) = q(\theta)q(D|\theta) \implies q(\theta|D) = \frac{q(D|\theta)}{q(D)}q(\theta). \quad (5.71)$$

Notice that this relates the conditional distribution $q(\theta|D)$ in terms of $q(D|\theta)$, $q(\theta)$ and $q(D)$, but that these are all different ways to write the same distribution. Bayes' rule does not tell us how we ought to *update* our beliefs in light of evidence, for that we need some other principle [Cat+11].

One of the nice things about this interpretation of Bayesian inference is that it naturally generalizes to other forms of constraints rather than assuming we have observed the data exactly.

If there was some additional measurement error that was well understood, we ought to instead of pegging out updated beliefs to be a delta function on the observed data, simply peg it to be the well understood distribution $p(D)$. For example, we might not know the precise value the data takes, but believe after measuring things that it is a Gaussian distribution with a certain mean and standard deviation.

Because of the chain rule of KL, this has no effect on our updated conditional distribution over parameters, which remains the Bayesian posterior: $p(\theta|D) = q(\theta|D)$. However, this does change our marginal beliefs about the parameters, which are now:

$$p(\theta) = \int dD p(D)q(\theta|D). \quad (5.72)$$

This generalization of Bayes' rule is sometimes called **Jeffrey's conditionalization rule** [Cat08].

5.1.8 KL divergence and exponential families

The KL divergence between two exponential family distributions from the same family has a nice closed form, as we explain below.

Consider $p(\mathbf{x})$ with natural parameter $\boldsymbol{\eta}$, base measure $h(\mathbf{x})$ and sufficient statistics $\mathcal{T}(\mathbf{x})$:

$$p(\mathbf{x}) = h(\mathbf{x}) \exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})] \quad (5.73)$$

where

$$A(\boldsymbol{\eta}) = \log \int h(\mathbf{x}) \exp(\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x})) d\mathbf{x} \quad (5.74)$$

is the *log partition function*, a convex function of $\boldsymbol{\eta}$.

The KL divergence between two exponential family distributions from the same family is as follows:

$$D_{\text{KL}}(p(\mathbf{x}|\boldsymbol{\eta}_1) \| p(\mathbf{x}|\boldsymbol{\eta}_2)) = \mathbb{E}_{\boldsymbol{\eta}_1} [(\boldsymbol{\eta}_1 - \boldsymbol{\eta}_2)^T \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta}_1) + A(\boldsymbol{\eta}_2)] \quad (5.75)$$

$$= (\boldsymbol{\eta}_1 - \boldsymbol{\eta}_2)^T \boldsymbol{\mu}_1 - A(\boldsymbol{\eta}_1) + A(\boldsymbol{\eta}_2) \quad (5.76)$$

where $\boldsymbol{\mu}_j \triangleq \mathbb{E}_{\boldsymbol{\eta}_j} [\mathcal{T}(\mathbf{x})]$.

5.1.8.1 Example: KL divergence between two Gaussians

An important example is the KL divergence between two multivariate Gaussian distributions, which is given by

$$\begin{aligned} D_{\text{KL}}(\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \| \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)) \\ = \frac{1}{2} \left[\text{tr}(\boldsymbol{\Sigma}_2^{-1} \boldsymbol{\Sigma}_1) + (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}_2^{-1} (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) - D + \log \left(\frac{\det(\boldsymbol{\Sigma}_2)}{\det(\boldsymbol{\Sigma}_1)} \right) \right] \end{aligned} \quad (5.77)$$

In the scalar case, this becomes

$$D_{\text{KL}}(\mathcal{N}(x|\mu_1, \sigma_1) \parallel \mathcal{N}(x|\mu_2, \sigma_2)) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \quad (5.78)$$

5.1.9 Approximating KL divergence using the Fisher information matrix

Let $p_{\boldsymbol{\theta}}(\mathbf{x})$ and $p_{\boldsymbol{\theta}'}(\mathbf{x})$ be two distributions, where $\boldsymbol{\theta}' = \boldsymbol{\theta} + \boldsymbol{\delta}$. We can measure how close the second distribution is to the first in terms their predictive distribution (as opposed to comparing $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ in parameter space) as follows:

$$D_{\text{KL}}(p_{\boldsymbol{\theta}} \parallel p_{\boldsymbol{\theta}'}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}) - \log p_{\boldsymbol{\theta}'}(\mathbf{x})] \quad (5.79)$$

Let us approximate this with a second order Taylor series expansion:

$$D_{\text{KL}}(p_{\boldsymbol{\theta}} \parallel p_{\boldsymbol{\theta}'}) \approx -\boldsymbol{\delta}^T \mathbb{E}[\nabla \log p_{\boldsymbol{\theta}}(\mathbf{x})] - \frac{1}{2} \boldsymbol{\delta}^T \mathbb{E}[\nabla^2 \log p_{\boldsymbol{\theta}}(\mathbf{x})] \boldsymbol{\delta} \quad (5.80)$$

Since the expected score function is zero (from Equation (3.44)), the first term vanishes, so we have

$$D_{\text{KL}}(p_{\boldsymbol{\theta}} \parallel p_{\boldsymbol{\theta}'}) \approx \frac{1}{2} \boldsymbol{\delta}^T \mathbf{F}(\boldsymbol{\theta}) \boldsymbol{\delta} \quad (5.81)$$

where \mathbf{F} is the FIM

$$\mathbf{F} = -\mathbb{E}[\nabla^2 \log p_{\boldsymbol{\theta}}(\mathbf{x})] = \mathbb{E}[(\nabla \log p_{\boldsymbol{\theta}}(\mathbf{x}))(\nabla \log p_{\boldsymbol{\theta}}(\mathbf{x}))^T] \quad (5.82)$$

Thus we have shown that the KL divergence is approximately equal to the (squared) Mahalanobis distance using the Fisher information matrix as the metric. This result is the basis of the **natural gradient** method discussed in Section 6.4.

5.1.10 Bregman divergence

Let $f : \Omega \rightarrow \mathbb{R}$ be a continuously differentiable, strictly convex function defined on a closed convex set Ω . We define the **Bregman divergence** associated with f as follows [Bre67]:

$$B_f(\mathbf{w} \parallel \mathbf{v}) = f(\mathbf{w}) - f(\mathbf{v}) - (\mathbf{w} - \mathbf{v})^T \nabla f(\mathbf{v}) \quad (5.83)$$

To understand this, let

$$\hat{f}_v(\mathbf{w}) = f(\mathbf{v}) + (\mathbf{w} - \mathbf{v})^T \nabla f(\mathbf{v}) \quad (5.84)$$

be a first order Taylor series approximation to f centered at \mathbf{v} . Then the Bregman divergence is the difference from this linear approximation:

$$B_f(\mathbf{w} \parallel \mathbf{v}) = f(\mathbf{w}) - \hat{f}_v(\mathbf{w}) \quad (5.85)$$

See Figure 5.3a for an illustration. Since f is convex, we have $B_f(\mathbf{w} \parallel \mathbf{v}) \geq 0$, since \hat{f}_v is a linear lower bound on f .

Below we mention some important special cases of Bregman divergences.

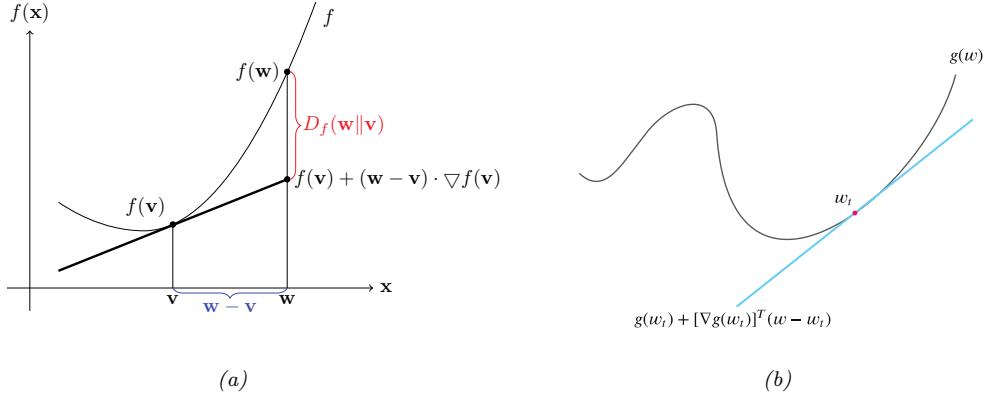


Figure 5.3: (a) Illustration of Bregman divergence. (b) A locally linear approximation to a non-convex function.

- If $f(\mathbf{w}) = \|\mathbf{w}\|^2$, then $B_f(\mathbf{w}||\mathbf{v}) = \|\mathbf{w} - \mathbf{v}\|^2$ is the squared Euclidean distance.
- If $f(\mathbf{w}) = \mathbf{w}^\top \mathbf{Q} \mathbf{w}$, then $B_f(\mathbf{w}||\mathbf{v})$ is the squared Mahalanobis distance.
- If \mathbf{w} are the natural parameters of an exponential family distribution, and $f(\mathbf{w}) = \log Z(\mathbf{w})$ is the log normalizer, then the Bregman divergence is the same as the Kullback-Leibler divergence, as we show in Section 5.1.10.1.

5.1.10.1 KL is a Bregman divergence

Recall that the log partition function $A(\boldsymbol{\eta})$ is a convex function. We can therefore use it to define the Bregman divergence (Section 5.1.10) between the two distributions, p and q , as follows:

$$B_f(\boldsymbol{\eta}_q || \boldsymbol{\eta}_p) = A(\boldsymbol{\eta}_q) - A(\boldsymbol{\eta}_p) - (\boldsymbol{\eta}_q - \boldsymbol{\eta}_p)^\top \nabla_{\boldsymbol{\eta}_p} A(\boldsymbol{\eta}_p) \quad (5.86)$$

$$= A(\boldsymbol{\eta}_q) - A(\boldsymbol{\eta}_p) - (\boldsymbol{\eta}_q - \boldsymbol{\eta}_p)^\top \mathbb{E}_p [\mathcal{T}(\mathbf{x})] \quad (5.87)$$

$$= D_{\text{KL}}(p || q) \quad (5.88)$$

where we exploited the fact that the gradient of the log partition function computes the expected sufficient statistics as shown in Section 2.4.3.

In fact, the KL divergence is the only divergence that is both a Bregman divergence and an f -divergence (Section 2.7.1) [Ama09].

5.2 Entropy

In this section, we discuss the **entropy** of a distribution p , which is just a shifted and scaled version of the KL divergence between the probability distribution and the uniform distribution, as we will see.

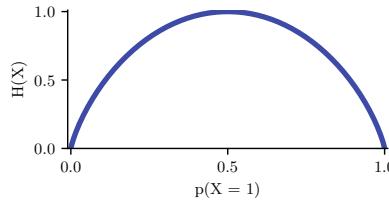


Figure 5.4: Entropy of a Bernoulli random variable as a function of θ . The maximum entropy is $\log_2 2 = 1$. Generated by [bernoulli_entropy_fig.ipynb](#).

5.2.1 Definition

The entropy of a discrete random variable X with distribution p over K states is defined by

$$\mathbb{H}(X) \triangleq -\sum_{k=1}^K p(X=k) \log p(X=k) = -\mathbb{E}_X [\log p(X)] \quad (5.89)$$

We can use logarithms to any base, but we commonly use log base 2, in which case the units are called bits, or log base e, in which case the units are called nats, as we explained in Section 5.1.3.1.

The entropy is equivalent to a constant minus the KL divergence from the uniform distribution:

$$\mathbb{H}(X) = \log K - D_{\text{KL}}(p(X) \parallel u(X)) \quad (5.90)$$

$$D_{\text{KL}}(p(X) \parallel u(X)) = \sum_{k=1}^K p(X=k) \log \frac{p(X=k)}{\frac{1}{K}} \quad (5.91)$$

$$= \log K + \sum_{k=1}^K p(X=k) \log p(X=k) \quad (5.92)$$

If p is uniform, the KL is zero, and we see that the entropy achieves its maximal value of $\log K$.

For the special case of binary random variables, $X \in \{0, 1\}$, we can write $p(X = 1) = \theta$ and $p(X = 0) = 1 - \theta$. Hence the entropy becomes

$$\mathbb{H}(X) = -[p(X=1) \log p(X=1) + p(X=0) \log p(X=0)] \quad (5.93)$$

$$= -[\theta \log \theta + (1 - \theta) \log(1 - \theta)] \quad (5.94)$$

This is called the **binary entropy function**, and is also written $\mathbb{H}(\theta)$. We plot this in Figure 5.4. We see that the maximum value of 1 bit occurs when the distribution is uniform, $\theta = 0.5$. A fair coin requires a single yes/no question to determine its state.

5.2.2 Differential entropy for continuous random variables

If X is a continuous random variable with pdf $p(x)$, we define the **differential entropy** as

$$h(X) \triangleq - \int_{\mathcal{X}} dx p(x) \log p(x) \quad (5.95)$$

assuming this integral exists.

For example, one can show that the entropy of a d -dimensional Gaussian is

$$h(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})) = \frac{1}{2} \log |2\pi e \boldsymbol{\Sigma}| = \frac{1}{2} \log[(2\pi e)^d |\boldsymbol{\Sigma}|] = \frac{d}{2} + \frac{d}{2} \log(2\pi) + \frac{1}{2} \log |\boldsymbol{\Sigma}| \quad (5.96)$$

In the 1d case, this becomes

$$h(\mathcal{N}(\mu, \sigma^2)) = \frac{1}{2} \log [2\pi e \sigma^2] \quad (5.97)$$

Note that, unlike the discrete case, *differential entropy can be negative*. This is because pdf's can be bigger than 1. For example, suppose $X \sim U(0, a)$. Then

$$h(X) = - \int_0^a dx \frac{1}{a} \log \frac{1}{a} = \log a \quad (5.98)$$

If we set $a = 1/8$, we have $h(X) = \log_2(1/8) = -3$ bits.

One way to understand differential entropy is to realize that all real-valued quantities can only be represented to finite precision. It can be shown [CT91, p228] that the entropy of an n -bit quantization of a continuous random variable X is approximately $h(X) + n$. For example, suppose $X \sim U(0, \frac{1}{8})$. Then in a binary representation of X , the first 3 bits to the right of the binary point must be 0 (since the number is $\leq 1/8$). So to describe X to n bits of accuracy only requires $n - 3$ bits, which agrees with $h(X) = -3$ calculated above.

The continuous entropy also lacks the reparameterization independence of KL divergence (Section 5.1.2.3). In particular, if we transform our random variable $y = f(x)$, the entropy transforms. To see this, note that the change of variables tells us that

$$p(y) dy = p(x) dx \implies p(y) = p(x) \left| \frac{dy}{dx} \right|^{-1}, \quad (5.99)$$

Thus the continuous entropy transforms as follows:

$$h(X) = - \int dx p(x) \log p(x) = h(Y) - \int dy p(y) \log \left| \frac{dy}{dx} \right|. \quad (5.100)$$

We pick up a factor in the continuous entropy of the log of the determinant of the Jacobian of the transformation. This changes the value for the continuous entropy even for simply rescaling the random variable such as when we change units. For example in Figure 5.5 we show the distribution of adult human heights (it is bimodal because while both male and female heights are normally distributed, they differ noticeably). The continuous entropy of this distribution depends on the units it is measured in. If measured in feet, the continuous entropy is 0.43 bits. Intuitively this is because human heights mostly span less than a foot. If measured in centimeters it is instead 5.4 bits. There are 30.48 centimeters in a foot, $\log_2 30.48 = 4.9$ explaining the difference. If we measured the continuous entropy of the same distribution measured in meters we would obtain -1.3 bits!

5.2.3 Typical sets

The **typical set** of a probability distribution is the set whose elements have an information content that is close to that of the expected information content from random samples from the distribution.

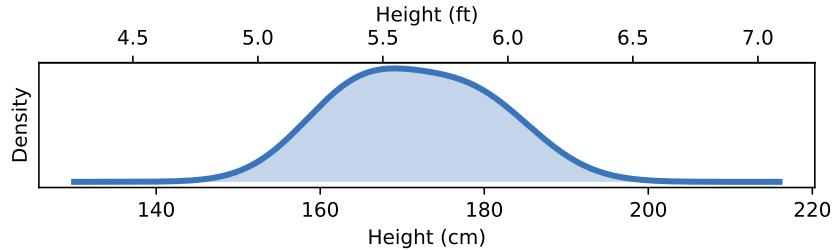


Figure 5.5: Distribution of adult heights. The continuous entropy of the distribution depends on its units of measurement. If heights are measured in feet, this distribution has a continuous entropy of 0.43 bits. If measured in centimeters it's 5.4 bits. If measured in meters it's -1.3 bits. Data taken from <https://ourworldindata.org/human-height>.

More precisely, for a distribution $p(\mathbf{x})$ with support $\mathbf{x} \in \mathcal{X}$, the ϵ -typical set $\mathcal{A}_\epsilon^N \in \mathcal{X}^N$ for $p(\mathbf{x})$ is the set of all length N sequences such that

$$\mathbb{H}(p(\mathbf{x})) - \epsilon \leq -\frac{1}{N} \log p(\mathbf{x}_1, \dots, \mathbf{x}_N) \leq \mathbb{H}(p(\mathbf{x})) + \epsilon \quad (5.101)$$

If we assume $p(\mathbf{x}_1, \dots, \mathbf{x}_N) = \prod_{n=1}^N p(\mathbf{x}_n)$, then we can interpret the term in the middle as the N -sample empirical estimate of the entropy. The **asymptotic equipartition property** or **AEP** states that this will converge (in probability) to the true entropy as $N \rightarrow \infty$ [CT06]. Thus the typical set has probability close to 1, and is thus a compact summary of what we can expect to be generated by $p(\mathbf{x})$.

5.2.4 Cross entropy and perplexity

A standard way to measure how close a model q is to a true distribution p is in terms of the KL divergence (Section 5.1), given by

$$D_{\text{KL}}(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{H}_{ce}(p, q) - \mathbb{H}(p) \quad (5.102)$$

where $\mathbb{H}_{ce}(p, q)$ is the **cross entropy**

$$\mathbb{H}_{ce}(p, q) = - \sum_x p(x) \log q(x) \quad (5.103)$$

and $\mathbb{H}(p) = \mathbb{H}_{ce}(p, p)$ is the entropy, which is a constant independent of the model.

In language modeling, it is common to report an alternative performance measure known as the **perplexity**. This is defined as

$$\text{perplexity}(p, q) \triangleq 2^{\mathbb{H}_{ce}(p, q)} \quad (5.104)$$

We can compute an empirical approximation to the cross entropy as follows. Suppose we approximate the true distribution with an empirical distribution based on data sampled from p :

$$p_{\mathcal{D}}(x|\mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \mathbb{I}(x = x_n) \quad (5.105)$$

In this case, the cross entropy is given by

$$H = -\frac{1}{N} \sum_{n=1}^N \log p(x_n) = -\frac{1}{N} \log \prod_{n=1}^N p(x_n) \quad (5.106)$$

The corresponding perplexity is given by

$$\text{perplexity}(p_{\mathcal{D}}, p) = 2^{-\frac{1}{N} \log(\prod_{n=1}^N p(x_n))} = 2^{\log(\prod_{n=1}^N p(x_n))^{-\frac{1}{N}}} \quad (5.107)$$

$$= (\prod_{n=1}^N p(x_n))^{-1/N} = \sqrt[N]{\prod_{n=1}^N \frac{1}{p(x_n)}} \quad (5.108)$$

In the case of language models, we usually condition on previous words when predicting the next word. For example, in a bigram model, we use a second order Markov model of the form $p(x_n|x_{n-1})$. We define the **branching factor** of a language model as the number of possible words that can follow any given word. For example, suppose the model predicts that each word is equally likely, regardless of context, so $p(x_n|x_{n-1}) = 1/K$, where K is the number of words in the vocabulary. Then the perplexity is $((1/K)^N)^{-1/N} = K$. If some symbols are more likely than others, and the model correctly reflects this, its perplexity will be lower than K . However, we have $\mathbb{H}(p^*) \leq \mathbb{H}_{ce}(p^*, p)$, so we can never reduce the perplexity below $2^{-\mathbb{H}(p^*)}$.

5.3 Mutual information

The KL divergence gave us a way to measure how similar two distributions were. How should we measure how dependent two random variables are? One thing we could do is turn the question of measuring the dependence of two random variables into a question about the similarity of their distributions. This gives rise to the notion of **mutual information** (MI) between two random variables, which we define below.

5.3.1 Definition

The mutual information between rv's X and Y is defined as follows:

$$\mathbb{I}(X; Y) \triangleq D_{\text{KL}}(p(x, y) \| p(x)p(y)) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \quad (5.109)$$

(We write $\mathbb{I}(X; Y)$ instead of $\mathbb{I}(X, Y)$, in case X and/or Y represent sets of variables; for example, we can write $\mathbb{I}(X; Y, Z)$ to represent the MI between X and (Y, Z) .) For continuous random variables, we just replace sums with integrals.

It is easy to see that MI is always non-negative, even for continuous random variables, since

$$\mathbb{I}(X; Y) = D_{\text{KL}}(p(x, y) \parallel p(x)p(y)) \geq 0 \quad (5.110)$$

We achieve the bound of 0 iff $p(x, y) = p(x)p(y)$.

5.3.2 Interpretation

Knowing that the mutual information is a KL divergence between the joint and factored marginal distributions tells us that the MI measures the information gain if we update from a model that treats the two variables as independent $p(x)p(y)$ to one that models their true joint density $p(x, y)$.

To gain further insight into the meaning of MI, it helps to re-express it in terms of joint and conditional entropies, as follows:

$$\mathbb{I}(X; Y) = \mathbb{H}(X) - \mathbb{H}(X|Y) = \mathbb{H}(Y) - \mathbb{H}(Y|X) \quad (5.111)$$

Thus we can interpret the MI between X and Y as the reduction in uncertainty about X after observing Y , or, by symmetry, the reduction in uncertainty about Y after observing X . Incidentally, this result gives an alternative proof that conditioning, on average, reduces entropy. In particular, we have $0 \leq \mathbb{I}(X; Y) = \mathbb{H}(X) - \mathbb{H}(X|Y)$, and hence $\mathbb{H}(X|Y) \leq \mathbb{H}(X)$.

We can also obtain a different interpretation. One can show that

$$\mathbb{I}(X; Y) = \mathbb{H}(X, Y) - \mathbb{H}(X|Y) - \mathbb{H}(Y|X) \quad (5.112)$$

Finally, one can show that

$$\mathbb{I}(X; Y) = \mathbb{H}(X) + \mathbb{H}(Y) - \mathbb{H}(X, Y) \quad (5.113)$$

See Figure 5.6 for a summary of these equations in terms of an **information diagram**. (Formally, this is a signed measure mapping set expressions to their information-theoretic counterparts [Yeu91a].)

5.3.3 Data processing inequality

Suppose we have an unknown variable X , and we observe a noisy function of it, call it Y . If we process the noisy observations in some way to create a new variable Z , it should be intuitively obvious that we cannot increase the amount of information we have about the unknown quantity, X . This is known as the **data processing inequality**. We now state this more formally, and then prove it.

Theorem 5.3.1. Suppose $X \rightarrow Y \rightarrow Z$ forms a Markov chain, so that $X \perp Z|Y$. Then $\mathbb{I}(X; Y) \geq \mathbb{I}(X; Z)$.

Proof. By the chain rule for mutual information we can expand the mutual information in two different ways:

$$\mathbb{I}(X; Y, Z) = \mathbb{I}(X; Z) + \mathbb{I}(X; Y|Z) \quad (5.114)$$

$$= \mathbb{I}(X; Y) + \mathbb{I}(X; Z|Y) \quad (5.115)$$

Since $X \perp Z|Y$, we have $\mathbb{I}(X; Z|Y) = 0$, so

$$\mathbb{I}(X; Z) + \mathbb{I}(X; Y|Z) = \mathbb{I}(X; Y) \quad (5.116)$$

Since $\mathbb{I}(X; Y|Z) \geq 0$, we have $\mathbb{I}(X; Y) \geq \mathbb{I}(X; Z)$. Similarly one can prove that $\mathbb{I}(Y; Z) \geq \mathbb{I}(X; Z)$. \square

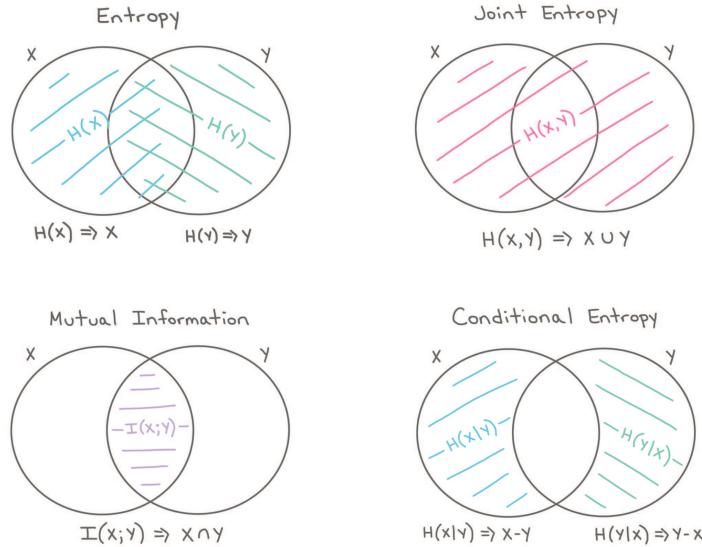


Figure 5.6: The marginal entropy, joint entropy, conditional entropy, and mutual information represented as information diagrams. Used with kind permission of Katie Everett.

5.3.4 Sufficient statistics

An important consequence of the DPI is the following. Suppose we have the chain $\theta \rightarrow X \rightarrow s(X)$. Then

$$\mathbb{I}(\theta; s(X)) \leq \mathbb{I}(\theta; X) \quad (5.117)$$

If this holds with equality, then we say that $s(X)$ is a **sufficient statistic** of the data X for the purposes of inferring θ . In this case, we can equivalently write $\theta \rightarrow s(X) \rightarrow X$, since we can reconstruct the data from knowing $s(X)$ just as accurately as from knowing θ .

An example of a sufficient statistic is the data itself, $s(X) = X$, but this is not very useful, since it doesn't summarize the data at all. Hence we define a **minimal sufficient statistic** $s(X)$ as one which is sufficient, and which contains no extra information about θ ; thus $s(X)$ maximally compresses the data X without losing information which is relevant to predicting θ . More formally, we say s is a minimal sufficient statistic for X if $s(X) = f(s'(X))$ for some function f and all sufficient statistics $s'(X)$. We can summarize the situation as follows:

$$\theta \rightarrow s(X) \rightarrow s'(X) \rightarrow X \quad (5.118)$$

Here $s'(X)$ takes $s(X)$ and adds redundant information to it, thus creating a one-to-many mapping.

For example, a minimal sufficient statistic for a set of N Bernoulli trials is simply N and $N_1 = \sum_n \mathbb{I}(X_n = 1)$, i.e., the number of successes. In other words, we don't need to keep track of the entire sequence of heads and tails and their ordering, we only need to keep track of the total number

of heads and tails. Similarly, for inferring the mean of a Gaussian distribution with known variance we only need to know the empirical mean and number of samples.

Earlier in Section 5.1.8 we motivated the exponential family of distributions as being the ones that are minimal in the sense that they contain no other information than constraints on some statistics of the data. It makes sense then that the statistics used to generate exponential family distributions are sufficient. It also hints at the more remarkable fact of the **Pitman-Koopman-Darmois theorem**, which says that for any distribution whose domain is fixed, it is only the exponential family that admits sufficient statistics with bounded dimensionality as the number of samples increases [Dia88b].

5.3.5 Multivariate mutual information

There are several ways to generalize the idea of mutual information to a set of random variables as we discuss below.

5.3.5.1 Total correlation

The simplest way to define multivariate MI is to use the **total correlation** [Wat60] or **multi-information** [SV98], defined as

$$\text{TC}(\{X_1, \dots, X_D\}) \triangleq D_{\text{KL}} \left(p(\mathbf{x}) \parallel \prod_d p(x_d) \right) \quad (5.119)$$

$$= \sum_{\mathbf{x}} p(\mathbf{x}) \log \frac{p(\mathbf{x})}{\prod_{d=1}^D p(x_d)} = \sum_d \mathbb{H}(x_d) - \mathbb{H}(\mathbf{x}) \quad (5.120)$$

For example, for 3 variables, this becomes

$$\text{TC}(X, Y, Z) = \mathbb{H}(X) + \mathbb{H}(Y) + \mathbb{H}(Z) - \mathbb{H}(X, Y, Z) \quad (5.121)$$

where $\mathbb{H}(X, Y, Z)$ is the joint entropy

$$\mathbb{H}(X, Y, Z) = - \sum_x \sum_y \sum_z p(x, y, z) \log p(x, y, z) \quad (5.122)$$

One can show that the multi-information is always non-negative, and is zero iff $p(\mathbf{x}) = \prod_d p(x_d)$. However, this means the quantity is non-zero even if only a pair of variables interact. For example, if $p(X, Y, Z) = p(X, Y)p(Z)$, then the total correlation will be non-zero, even though there is no 3 way interaction. This motivates the alternative definition in Section 5.3.5.2.

5.3.5.2 Interaction information (co-information)

The conditional mutual information can be used to give an inductive definition of the **multivariate mutual information (MMI)** as follows:

$$\mathbb{I}(X_1; \dots; X_D) = \mathbb{I}(X_1; \dots; X_{D-1}) - \mathbb{I}(X_1; \dots; X_{D-1}|X_D) \quad (5.123)$$

This is called the **multiple mutual information** [Yeu91b], or the **co-information** [Bel03]. This definition is equivalent, up to a sign change, to the **interaction information** [McG54; Han80; JB03; Bro09].

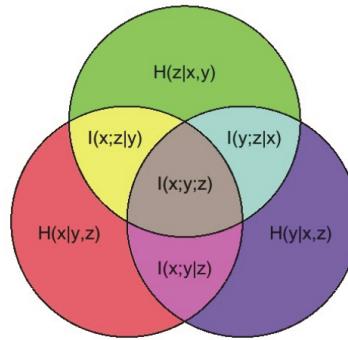


Figure 5.7: Illustration of multivariate mutual information between three random variables. From https://en.wikipedia.org/wiki/Mutual_information. Used with kind permission of Wikipedia author PAR.

For 3 variables, the MMI is given by

$$\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Y) - \mathbb{I}(X;Y|Z) \quad (5.124)$$

$$= \mathbb{I}(X;Z) - \mathbb{I}(X;Z|Y) \quad (5.125)$$

$$= \mathbb{I}(Y;Z) - \mathbb{I}(Y;Z|X) \quad (5.126)$$

This can be interpreted as the change in mutual information between two pairs of variables when conditioning on the third. Note that this quantity is symmetric in its arguments.

By the definition of conditional mutual information, we have

$$\mathbb{I}(X;Z|Y) = \mathbb{I}(Z;X,Y) - \mathbb{I}(Y;Z) \quad (5.127)$$

Hence we can rewrite Equation (5.125) as follows:

$$\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Z) + \mathbb{I}(Y;Z) - \mathbb{I}(X,Y;Z) \quad (5.128)$$

This tells us that the MMI is the difference between how much we learn about Z given X and Y individually vs jointly (see also Section 5.3.5.3).

The 3-way MMI is illustrated in the information diagram in Figure 5.7. The way to interpret such diagrams when we have multiple variables is as follows: the area of a shaded area that includes circles A, B, C, \dots and excludes circles F, G, H, \dots represents $\mathbb{I}(A;B;C;\dots|F,G,H,\dots)$; if $B = C = \emptyset$, this is just $\mathbb{H}(A|F,G,H,\dots)$; if $F = G = H = \emptyset$, this is just $\mathbb{I}(A;B;C,\dots)$.

5.3.5.3 Synergy and redundancy

The MMI is $\mathbb{I}(X;Y;Z) = \mathbb{I}(X;Z) + \mathbb{I}(Y;Z) - \mathbb{I}(X,Y;Z)$. We see that this can be positive, zero, or negative. If some of the information about Z that is provided by X is also provided by Y , then there is some **redundancy** between X and Y (wrt Z). In this case, $\mathbb{I}(X;Z) + \mathbb{I}(Y;Z) > \mathbb{I}(X,Y;Z)$, so (from Equation (5.128)) we see that the MMI will be positive. If, by contrast, we learn more about Z when we see X and Y together, we say there is some **synergy** between them. In this case, $\mathbb{I}(X;Z) + \mathbb{I}(Y;Z) < \mathbb{I}(X,Y;Z)$, so the MMI will be negative.

5.3.5.4 MMI and causality

The sign of the MMI can be used to distinguish between different kinds of directed graphical models, which can sometimes be interpreted causally (see Chapter 36 for a general discussion of causality). For example, consider a model of the form $X \leftarrow Z \rightarrow Y$, where Z is a “cause” of X and Y . For example, suppose X represents the event it is raining, Y represents the event that the sky is dark, and Z represents the event that the sky is cloudy. Conditioning on the common cause Z renders the children X and Y independent, since if I know it is cloudy, noticing that the sky is dark does not change my beliefs about whether it will rain or not. Consequently $\mathbb{I}(X; Y|Z) \leq \mathbb{I}(X; Y)$, so $\mathbb{I}(X; Y; Z) \geq 0$.

Now consider the case where Z is a common effect, $X \rightarrow Z \leftarrow Y$. In this case, conditioning on Z makes X and Y dependent, due to the explaining away phenomenon (see Section 4.2.4.2). For example, if X and Y are independent random bits, and Z is the XOR of X and Y , then observing $Z = 1$ means that $p(X \neq Y|Z = 1) = 1$, so X and Y are now dependent (information-theoretically, not causally), even though they were a priori independent. Consequently $\mathbb{I}(X; Y|Z) \geq \mathbb{I}(X; Y)$, so $\mathbb{I}(X; Y; Z) \leq 0$.

Finally, consider a Markov chain, $X \rightarrow Y \rightarrow Z$. We have $\mathbb{I}(X; Z|Y) \leq \mathbb{I}(X; Z)$ and so the MMI must be positive.

5.3.5.5 MMI and entropy

We can also write the MMI in terms of entropies. Specifically, we know that

$$\mathbb{I}(X; Y) = \mathbb{H}(X) + \mathbb{H}(Y) - \mathbb{H}(X, Y) \quad (5.129)$$

and

$$\mathbb{I}(X; Y|Z) = \mathbb{H}(X, Z) + \mathbb{H}(Y, Z) - \mathbb{H}(Z) - \mathbb{H}(X, Y, Z) \quad (5.130)$$

Hence we can rewrite Equation (5.124) as follows:

$$\mathbb{I}(X; Y; Z) = [\mathbb{H}(X) + \mathbb{H}(Y) + \mathbb{H}(Z)] - [\mathbb{H}(X, Y) + \mathbb{H}(X, Z) + \mathbb{H}(Y, Z)] + \mathbb{H}(X, Y, Z) \quad (5.131)$$

Contrast this to Equation (5.121).

More generally, we have

$$\mathbb{I}(X_1, \dots, X_D) = - \sum_{\mathcal{T} \subseteq \{1, \dots, D\}} (-1)^{|\mathcal{T}|} \mathbb{H}(\mathcal{T}) \quad (5.132)$$

For sets of size 1, 2, and 3 this expands as follows:

$$I_1 = H_1 \quad (5.133)$$

$$I_{12} = H_1 + H_2 - H_{12} \quad (5.134)$$

$$I_{123} = H_1 + H_2 + H_3 - H_{12} - H_{13} - H_{23} + H_{123} \quad (5.135)$$

We can use the **Möbius inversion formula** to derive the following dual relationship:

$$\mathbb{H}(\mathcal{S}) = - \sum_{\mathcal{T} \subseteq \mathcal{S}} (-1)^{|\mathcal{T}|} \mathbb{I}(\mathcal{T}) \quad (5.136)$$

for sets of variables \mathcal{S} .

Using the chain rule for entropy, we can also derive the following expression for the 3-way MMI:

$$\mathbb{I}(X; Y; Z) = \mathbb{H}(Z) - \mathbb{H}(Z|X) - \mathbb{H}(Z|Y) + \mathbb{H}(Z|X, Y) \quad (5.137)$$

5.3.6 Variational bounds on mutual information

In this section, we discuss methods for computing upper and lower bounds on MI that use variational approximations to the intractable distributions. This can be useful for representation learning (Chapter 32). This approach was first suggested in [BA03]. For a more detailed overview of variational bounds on mutual information, see Poole et al. [Poo+19b].

5.3.6.1 Upper bound

Suppose that the joint $p(\mathbf{x}, \mathbf{y})$ is intractable to evaluate, but that we can sample from $p(\mathbf{x})$ and evaluate the conditional distribution $p(\mathbf{y}|\mathbf{x})$. Furthermore, suppose we approximate $p(\mathbf{y})$ by $q(\mathbf{y})$. Then we can compute an upper bound on the MI as follows:

$$\mathbb{I}(\mathbf{x}; \mathbf{y}) = \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} \left[\log \frac{p(\mathbf{y}|\mathbf{x})q(\mathbf{y})}{p(\mathbf{y})q(\mathbf{y})} \right] \quad (5.138)$$

$$= \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} \left[\log \frac{p(\mathbf{y}|\mathbf{x})}{q(\mathbf{y})} \right] - D_{\text{KL}}(p(\mathbf{y}) \| q(\mathbf{y})) \quad (5.139)$$

$$\leq \mathbb{E}_{p(\mathbf{x})} \left[\mathbb{E}_{p(\mathbf{y}|\mathbf{x})} \left[\log \frac{p(\mathbf{y}|\mathbf{x})}{q(\mathbf{y})} \right] \right] \quad (5.140)$$

$$= \mathbb{E}_{p(\mathbf{x})} [D_{\text{KL}}(p(\mathbf{y}|\mathbf{x}) \| q(\mathbf{y}))] \quad (5.141)$$

This bound is tight if $q(\mathbf{y}) = p(\mathbf{y})$.

What's happening here is that $\mathbb{I}(Y; X) = \mathbb{H}(Y) - \mathbb{H}(Y|X)$ and we've assumed we know $p(\mathbf{y}|\mathbf{x})$ and so can estimate $\mathbb{H}(Y|X)$ well. While we don't know $\mathbb{H}(Y)$, we can upper bound it using some model $q(\mathbf{y})$. Our model can never do better than $p(\mathbf{y})$ itself (the non-negativity of KL), so our entropy estimate errs too large, and hence our MI estimate will be an upper bound.

5.3.6.2 BA lower bound

Suppose that the joint $p(\mathbf{x}, \mathbf{y})$ is intractable to evaluate, but that we can evaluate $p(\mathbf{x})$. Furthermore, suppose we approximate $p(\mathbf{x}|\mathbf{y})$ by $q(\mathbf{x}|\mathbf{y})$. Then we can derive the following variational lower bound on the mutual information:

$$\mathbb{I}(\mathbf{x}; \mathbf{y}) = \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} \left[\log \frac{p(\mathbf{x}|\mathbf{y})}{p(\mathbf{x})} \right] \quad (5.142)$$

$$= \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} \left[\log \frac{q(\mathbf{x}|\mathbf{y})}{p(\mathbf{x})} \right] + \mathbb{E}_{p(\mathbf{y})} [D_{\text{KL}}(p(\mathbf{x}|\mathbf{y}) \| q(\mathbf{x}|\mathbf{y}))] \quad (5.143)$$

$$\geq \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} \left[\log \frac{q(\mathbf{x}|\mathbf{y})}{p(\mathbf{x})} \right] = \mathbb{E}_{p(\mathbf{x}, \mathbf{y})} [\log q(\mathbf{x}|\mathbf{y})] + h(\mathbf{x}) \quad (5.144)$$

where $h(\mathbf{x})$ is the differential entropy of \mathbf{x} . This is called the **BA lower bound**, after the authors Barber and Agakov [BA03].

5.3.6.3 NWJ lower bound

The BA lower bound requires a tractable normalized distribution $q(\mathbf{x}|\mathbf{y})$ that we can evaluate pointwise. If we reparameterize this distribution in a clever way, we can generate a lower bound that does not require a normalized distribution. Let's write:

$$q(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x})e^{f(\mathbf{x},\mathbf{y})}}{Z(\mathbf{y})} \quad (5.145)$$

with $Z(\mathbf{y}) = \mathbb{E}_{p(\mathbf{x})} [e^{f(\mathbf{x},\mathbf{y})}]$ the normalization constant or partition function. Plugging this into the BA lower bound above we obtain:

$$\mathbb{E}_{p(\mathbf{x},\mathbf{y})} \left[\log \frac{p(\mathbf{x})e^{f(\mathbf{x},\mathbf{y})}}{p(\mathbf{x})Z(\mathbf{y})} \right] = \mathbb{E}_{p(\mathbf{x},\mathbf{y})} [f(\mathbf{x},\mathbf{y})] - \mathbb{E}_{p(\mathbf{y})} [\log Z(\mathbf{y})] \quad (5.146)$$

$$= \mathbb{E}_{p(\mathbf{x},\mathbf{y})} [f(\mathbf{x},\mathbf{y})] - \mathbb{E}_{p(\mathbf{y})} \left[\log \mathbb{E}_{p(\mathbf{x})} \left[e^{f(\mathbf{x},\mathbf{y})} \right] \right] \quad (5.147)$$

$$\triangleq I_{DV}(X;Y). \quad (5.148)$$

This is the **Donsker-Varadhan** lower bound [DV75].

We can construct a more tractable version of this by using the fact that the log function can be upper bounded by a straight line using

$$\log x \leq \frac{x}{a} + \log a - 1 \quad (5.149)$$

If we set $a = e$, we get

$$\mathbb{I}(X;Y) \geq \mathbb{E}_{p(\mathbf{x},\mathbf{y})} [f(\mathbf{x},\mathbf{y})] - e^{-1} \mathbb{E}_{p(\mathbf{y})} Z(\mathbf{y}) \triangleq I_{NWJ}(X;Y) \quad (5.150)$$

This is called the **NWJ lower bound** (after the authors of Nguyen, Wainwright, and Jordan [NWJ10a]), or the f-GAN KL [NCT16a], or the MINE-f score [Bel+18].

5.3.6.4 InfoNCE lower bound

If we instead explore a multi-sample extension to the DV bound above, we can generate the following lower bound (see [Poo+19b] for the derivation):

$$\mathbb{I}_{NCE} = \mathbb{E} \left[\frac{1}{K} \sum_{i=1}^K \log \frac{e^{f(\mathbf{x}_i,\mathbf{y}_i)}}{\frac{1}{K} \sum_{j=1}^K e^{f(\mathbf{x}_i,\mathbf{y}_j)}} \right] \quad (5.151)$$

$$= \log K - \mathbb{E} \left[\frac{1}{K} \sum_{i=1}^K \log \left(1 + \sum_{j \neq i}^K e^{f(\mathbf{x}_i,\mathbf{y}_j) - f(\mathbf{x}_i,\mathbf{y}_i)} \right) \right] \quad (5.152)$$

where the expectation is over paired samples from the joint $p(X,Y)$. The quantity in Equation (5.152) is called the **InfoNCE** estimate, and was proposed in [OLV18a; Hen+19a]. (NCE stands for “noise contrastive estimation”, and is discussed in Section 24.4.)

The intuition here is that mutual information is a divergence between the joint $p(\mathbf{x},\mathbf{y})$ and the product of the marginals, $p(\mathbf{x})p(\mathbf{y})$. In other words, mutual information is a measurement of how

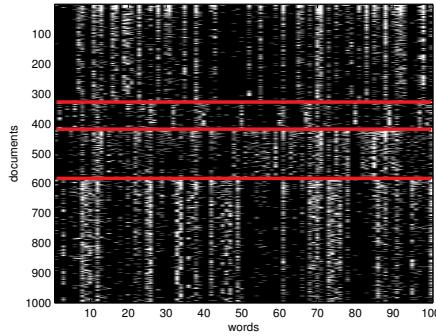


Figure 5.8: Subset of size 16242×100 of the 20-newsgroups data. We only show 1000 rows, for clarity. Each row is a document (represented as a bag-of-words bit vector), each column is a word. The red lines separate the 4 classes, which are (in descending order) comp, rec, sci, talk (these are the titles of USENET groups). We can see that there are subsets of words whose presence or absence is indicative of the class. The data is available from <http://cs.nyu.edu/~roweis/data.html>. Generated by `newsgroups_visualize.ipynb`.

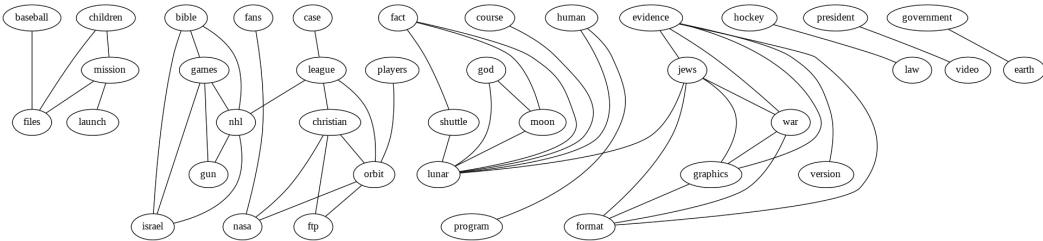


Figure 5.9: Part of a relevance network constructed from the 20-newsgroup data. data shown in Figure 5.8. We show edges whose mutual information is greater than or equal to 20% of the maximum pairwise MI. For clarity, the graph has been cropped, so we only show a subset of the nodes and edges. Generated by `relevance_network_newsgroup_demo.ipynb`.

distinct sampling pairs jointly is from sampling \mathbf{x} s and \mathbf{y} s independently. The InfoNCE bound in Equation (5.152) provides a lower bound on the true mutual information by attempting to train a model to distinguish between these two situations.

Although this is a valid lower bound, we may need to use a large batch size K to estimate the MI if the MI is large, since $\mathbb{I}_{\text{NCE}} \leq \log K$. (Recently [SE20a] proposed to use a multi-label classifier, rather than a multi-class classifier, to overcome this limitation.)

5.3.7 Relevance networks

If we have a set of related variables, we can compute a **relevance network**, in which we add an $i - j$ edge if the pairwise mutual information $\mathbb{I}(X_i; X_j)$ is above some threshold. In the Gaussian case, $\mathbb{I}(X_i; X_j) = -\frac{1}{2} \log(1 - \rho_{ij}^2)$, where ρ_{ij} is the correlation coefficient, and the resulting graph is called a **covariance graph** (Section 4.5.5.1). However, we can also apply it to discrete random variables.

Relevance networks are quite popular in systems biology [Mar+06], where they are used to visualize

the interaction between genes. But they can also be applied to other kinds of datasets. For example, Figure 5.9 visualizes the MI between words in the 20-newsgroup dataset shown in Figure 5.8. The results seem intuitively reasonable.

However, relevance networks suffer from a major problem: the graphs are usually very dense, since most variables are dependent on most other variables, even after thresholding the MIs. For example, suppose X_1 directly influences X_2 which directly influences X_3 (e.g., these form components of a signalling cascade, $X_1 - X_2 - X_3$). Then X_1 has non-zero MI with X_3 (and vice versa), so there will be a $1 - 3$ edge as well as the $1 - 2$ and $2 - 3$ edges; thus the graph may be fully connected, depending on the threshold.

A solution to this is to learn a probabilistic graphical model, which represents conditional *independence*, rather than *dependence*. In the chain example, there will not be a $1 - 3$ edge, since $X_1 \perp X_3 | X_2$. Consequently graphical models are usually much sparser than relevance networks. See Chapter 30 for details.

5.4 Data compression (source coding)

Data compression, also known as **source coding**, is at the heart of information theory. It is also related to probabilistic machine learning. The reason for this is as follows: if we can model the probability of different kinds of data samples, then we can assign short **code words** to the most frequently occurring ones, reserving longer encodings for the less frequent ones. This is similar to the situation in natural language, where common words (such as “a”, “the”, “and”) are generally much shorter than rare words. Thus the ability to compress data requires an ability to discover the underlying patterns, and their relative frequencies, in the data. This has led Marcus Hutter to propose that compression be used as an objective way to measure performance towards general purpose AI. More precisely, he is offering 50,000 Euros to anyone who can compress the first 100MB of (English) Wikipedia better than some baseline. This is known as the **Hutter prize**.¹

In this section, we give a brief summary of some of the key ideas in data compression. For details, see e.g., [Mac03; CT06; YMT22].

5.4.1 Lossless compression

Discrete data, such as natural language, can always be compressed in such a way that we can uniquely recover the original data. This is called **lossless compression**.

Claude Shannon proved that the expected number of bits needed to losslessly encode some data coming from distribution p is at least $\mathbb{H}(p)$. This is known as the **source coding theorem**. Achieving this lower bound requires coming up with good probability models, as well as good ways to design codes based on those models. Because of the non-negativity of the KL divergence, $\mathbb{H}_{ce}(p, q) \geq \mathbb{H}(p)$, so if we use any model q other than the true model p to compress the data, it will take some excess bits. The number of excess bits is exactly $D_{\text{KL}}(p \| q)$.

Common techniques for realizing lossless codes include Huffman coding, arithmetic coding, and asymmetric numeral systems [Dud13]. The input to these algorithms is a probability distribution over strings (which is where ML comes in). This distribution is often represented using a latent variable model (see e.g., [TBB19; KAH19]).

1. For details, see <http://prize.hutter1.net>.

5.4.2 Lossy compression and the rate-distortion tradeoff

To encode real-valued signals, such as images and sound, as a digital signal, we first have to quantize the signal into a sequence of symbols. A simple way to do this is to use vector quantization. We can then compress this discrete sequence of symbols using lossless coding methods. However, when we uncompress, we lose some information. Hence this approach is called **lossy compression**.

In this section, we quantify this tradeoff between the size of the representation (number of symbols we use), and the resulting error. We will use the terminology of the variational information bottleneck discussed in Section 5.6.2 (except here we are in the unsupervised setting). In particular, we assume we have a stochastic encoder $p(\mathbf{z}|\mathbf{x})$, a stochastic decoder $d(\mathbf{x}|\mathbf{z})$ and a prior marginal $m(\mathbf{z})$.

We define the **distortion** of an encoder-decoder pair (as in Section 5.6.2) as follows:

$$D = - \int d\mathbf{x} p(\mathbf{x}) \int d\mathbf{z} e(\mathbf{z}|\mathbf{x}) \log d(\mathbf{x}|\mathbf{z}) \quad (5.153)$$

If the decoder is a deterministic model plus Gaussian noise, $d(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|f_d(\mathbf{z}), \sigma^2)$, and the encoder is deterministic, $e(\mathbf{z}|\mathbf{x}) = \delta(\mathbf{z} - f_e(\mathbf{x}))$, then this becomes

$$D = \frac{1}{\sigma^2} \mathbb{E}_{p(\mathbf{x})} [| | f_d(f_e(\mathbf{x})) - \mathbf{x} | |^2] \quad (5.154)$$

This is just the expected **reconstruction error** that occurs if we (deterministically) encode and then decode the data using f_e and f_d .

We define the **rate** of our model as follows:

$$R = \int d\mathbf{x} p(\mathbf{x}) \int d\mathbf{z} e(\mathbf{z}|\mathbf{x}) \log \frac{e(\mathbf{z}|\mathbf{x})}{m(\mathbf{z})} \quad (5.155)$$

$$= \mathbb{E}_{p(\mathbf{x})} [D_{\text{KL}}(e(\mathbf{z}|\mathbf{x}) \parallel m(\mathbf{z}))] \quad (5.156)$$

$$= \int d\mathbf{x} \int d\mathbf{z} p(\mathbf{x}, \mathbf{z}) \log \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{x})m(\mathbf{z})} \geq I(\mathbf{x}, \mathbf{z}) \quad (5.157)$$

This is just the average KL between our encoding distribution and the marginal. If we use $m(\mathbf{z})$ to design an optimal code, then the rate is the *excess* number of bits we need to pay to encode our data using $m(\mathbf{z})$ rather than the true **aggregate posterior** $p(\mathbf{z}) = \int d\mathbf{x} p(\mathbf{x})e(\mathbf{z}|\mathbf{x})$.

There is a fundamental tradeoff between the rate and distortion. To see why, note that a trivial encoding scheme would set $e(\mathbf{z}|\mathbf{x}) = \delta(\mathbf{z} - \mathbf{x})$, which simply uses \mathbf{x} as its own best representation. This would incur 0 distortion (and hence maximize the likelihood), but it would incur a high rate, since each $e(\mathbf{z}|\mathbf{x})$ distribution would be unique, and far from $m(\mathbf{z})$. In other words, there would be no compression. Conversely, if $e(\mathbf{z}|\mathbf{x}) = \delta(\mathbf{z} - \mathbf{0})$, the encoder would ignore the input. In this case, the rate would be 0, but the distortion would be high.

We can characterize the tradeoff more precisely using the variational lower and upper bounds on the mutual information from Section 5.3.6. From that section, we know that

$$H - D \leq I(\mathbf{x}; \mathbf{z}) \leq R \quad (5.158)$$

where H is the (differential) entropy

$$H = - \int d\mathbf{x} p(\mathbf{x}) \log p(\mathbf{x}) \quad (5.159)$$

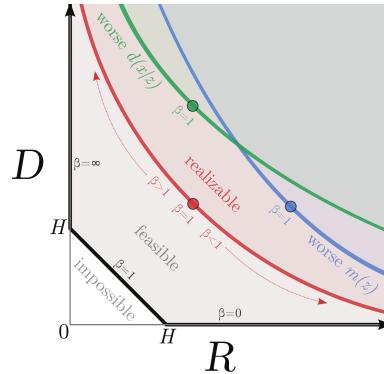


Figure 5.10: Illustration of the rate-distortion tradeoff. See text for details. From Figure 1 of [Ale+18]. Used with kind permission of Alex Alemi.

For discrete data, all probabilities are bounded above by 1, and hence $H \geq 0$ and $D \geq 0$. In addition, the rate is always non-negative, $R \geq 0$, since it is the average of a KL divergence. (This is true for either discrete or continuous encodings \mathbf{z} .) Consequently, we can plot the set of achievable values of R and D as shown in Figure 5.10. This is known as a **rate distortion curve**.

The bottom horizontal line corresponds to the zero distortion setting, $D = 0$, in which we can perfectly encode and decode our data. This can be achieved by using the trivial encoder where $e(\mathbf{z}|\mathbf{x}) = \delta(\mathbf{z} - \mathbf{x})$. Shannon's source coding theorem tells us that the minimum number of bits we need to use to encode data in this setting is the entropy of the data, so $R \geq H$ when $D = 0$. If we use a suboptimal marginal distribution $m(\mathbf{z})$ for coding, we will increase the rate without affecting the distortion.

The left vertical line corresponds to the zero rate setting, $R = 0$, in which the latent code is independent of \mathbf{z} . In this case, the decoder $d(\mathbf{x}|\mathbf{z})$ is independent of \mathbf{z} . However, we can still learn a joint probability model $p(\mathbf{x})$ which does not use latent variables, e.g., this could be an autoregressive model. The minimal distortion such a model could achieve is again the entropy of the data, $D \geq H$.

The black diagonal line illustrates solutions that satisfy $D = H - R$, where the upper and lower bounds are tight. In practice, we cannot achieve points on the diagonal, since that requires the bounds to be tight, and therefore assumes our models $e(\mathbf{z}|\mathbf{x})$ and $d(\mathbf{x}|\mathbf{z})$ are perfect. This is called the “non-parametric limit”. In the finite data setting, we will always incur additional error, so the RD plot will trace a curve which is shifted up, as shown in Figure 5.10.

We can generate different solutions along this curve by minimizing the following objective:

$$J = D + \beta R = \int d\mathbf{x} p(\mathbf{x}) \int d\mathbf{z} e(\mathbf{z}|\mathbf{x}) \left[-\log d(\mathbf{x}|\mathbf{z}) + \beta \log \frac{e(\mathbf{z}|\mathbf{x})}{m(\mathbf{z})} \right] \quad (5.160)$$

If we set $\beta = 1$, and define $q(\mathbf{z}|\mathbf{x}) = e(\mathbf{z}|\mathbf{x})$, $p(\mathbf{x}|\mathbf{z}) = d(\mathbf{x}|\mathbf{z})$, and $p(\mathbf{z}) = m(\mathbf{z})$, this exactly matches the VAE objective in Section 21.2. To see this, note that the ELBO from Section 10.1.1.2 can be written as

$$\mathcal{L} = -(D + R) = \mathbb{E}_{p(\mathbf{x})} \left[\mathbb{E}_{e(\mathbf{z}|\mathbf{x})} [\log d(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{e(\mathbf{z}|\mathbf{x})} \left[\log \frac{e(\mathbf{z}|\mathbf{x})}{m(\mathbf{z})} \right] \right] \quad (5.161)$$

which we recognize as the expected reconstruction error minus the KL term $D_{\text{KL}}(e(\mathbf{z}|\mathbf{x}) \parallel m(\mathbf{z}))$.

If we allow $\beta \neq 1$, we recover the β -VAE objective discussed in Section 21.3.1. Note, however, that the β -VAE model cannot distinguish between different solutions on the diagonal line, all of which have $\beta = 1$. This is because all such models have the same marginal likelihood (and hence same ELBO), although they differ radically in terms of whether they learn an interesting latent representation or not. Thus likelihood is not a sufficient metric for comparing the quality of unsupervised representation learning methods, as discussed in Section 21.3.1.

For further discussion on the inherent conflict between rate, distortion, and *perception*, see [BM19]. For techniques for evaluating rate distortion curves for models see [HCG20].

5.4.3 Bits back coding

In the previous section we penalized the rate of our code using the average KL divergence, $\mathbb{E}_{p(\mathbf{x})}[R(\mathbf{x})]$, where

$$R(\mathbf{x}) \triangleq \int d\mathbf{z} p(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{z}|\mathbf{x})}{m(\mathbf{z})} = \mathbb{H}_{ce}(p(\mathbf{z}|\mathbf{x}), m(\mathbf{z})) - \mathbb{H}(p(\mathbf{z}|\mathbf{x})). \quad (5.162)$$

The first term is the cross entropy, which is the expected number of bits we need to encode \mathbf{x} ; the second term is the entropy, which is the minimum number of bits. Thus we are penalizing the *excess* number of bits required to communicate the code to a receiver. How come we don't have to "pay for" the actual (total) number of bits we use, which is the cross entropy?

The reason is that we could in principle get the bits needed by the optimal code given back to us; this is called **bits back coding** [HC93; FH97]. The argument goes as follows. Imagine Alice is trying to (losslessly) communicate some data, such as an image \mathbf{x} , to Bob. Before they went their separate ways, both Alice and Bob decided to share their encoder $p(\mathbf{z}|\mathbf{x})$, marginal $m(\mathbf{z})$ and decoder distributions $d(\mathbf{x}|\mathbf{z})$. To communicate an image, Alice will use a **two part code**. First, she will sample a code $\mathbf{z} \sim p(\mathbf{z}|\mathbf{x})$ from her encoder, and communicate that to Bob over a channel designed to efficiently encode samples from the marginal $m(\mathbf{z})$; this costs $-\log_2 m(\mathbf{z})$ bits. Next Alice will use her decoder $d(\mathbf{x}|\mathbf{z})$ to compute the residual error, and losslessly send that to Bob at the cost of $-\log_2 d(\mathbf{x}|\mathbf{z})$ bits. The expected total number of bits required here is what we naively expected:

$$\mathbb{E}_{p(\mathbf{z}|\mathbf{x})}[-\log_2 d(\mathbf{x}|\mathbf{z}) - \log_2 m(\mathbf{z})] = D + \mathbb{H}_{ce}(p(\mathbf{z}|\mathbf{x}), m(\mathbf{z})). \quad (5.163)$$

We see that this is the distortion plus cross entropy, not distortion plus rate. So how do we get the bits back, to convert the cross entropy to a rate term?

The trick is that Bob actually receives more information than we suspected. Bob can use the code \mathbf{z} and the residual error to perfectly reconstruct \mathbf{x} . However, Bob also knows what specific code Alice sent, \mathbf{z} , as well as what encoder she used, $p(\mathbf{z}|\mathbf{x})$. When Alice drew the sample code $\mathbf{z} \sim p(\mathbf{z}|\mathbf{x})$, she had to use some kind of entropy source in order to generate the random sample. Suppose she did it by picking words sequentially from a compressed copy of Moby Dick, in order to generate a stream of random bits. On Bob's end, he can reverse engineer all of the sampling bits, and thus recover the compressed copy of Moby Dick! Thus Alice can use the extra randomness in the choice of \mathbf{z} to share more information.

While in the original formulation the bits back argument was largely theoretical, offering a thought experiment for why we should penalize our models with the KL instead of the cross entropy, recently several practical real world algorithms have been developed that actually achieve the bits back goal. These include [PHL19; AT20; TBB19; YBM20; HLA19; FHL20].

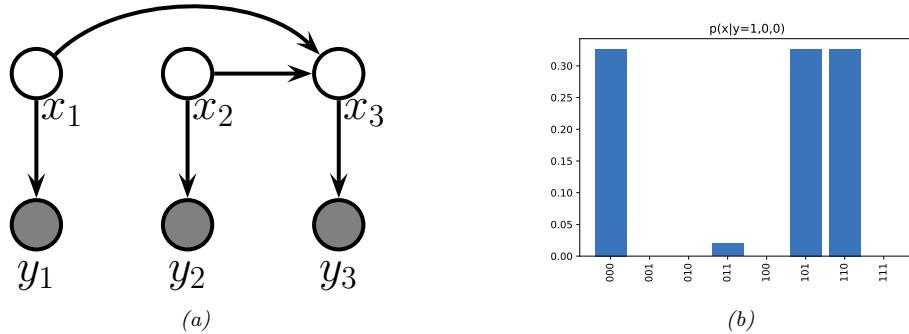


Figure 5.11: (a) A simple error-correcting code DPGM. x_i are the sent bits, y_i are the received bits. x_3 is an even parity check bit computed from x_1 and x_2 . (b) Posterior over codewords given that $\mathbf{y} = (1, 0, 0)$; the probability of a bit flip is 0.2. Generated by [error_correcting_code_demo.ipynb](#).

5.5 Error-correcting codes (channel coding)

The idea behind **error correcting codes** is to add redundancy to a signal \mathbf{x} (which is the result of encoding the original data), such that when it is sent over to the receiver via a noisy transmission line (such as a cell phone connection), the receiver can recover from any corruptions that might occur to the signal. This is called **channel coding**.

In more detail, let $\mathbf{x} \in \{0, 1\}^m$ be the source message, where m is called the **block length**. Let \mathbf{y} be the result of sending \mathbf{x} over a **noisy channel**. This is a corrupted version of the message. For example, each message bit may get flipped independently with probability α , in which case $p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^m p(y_i|x_i)$, where $p(y_i|x_i = 0) = [1 - \alpha, \alpha]$ and $p(y_i|x_i = 1) = [\alpha, 1 - \alpha]$. Alternatively, we may add Gaussian noise, so $p(y_i|x_i = b) = \mathcal{N}(y_i|\mu_b, \sigma^2)$. The receiver's goal is to infer the true message from the noisy observations, i.e., to compute $\text{argmax}_{\mathbf{x}} p(\mathbf{x}|\mathbf{y})$.

A common way to increase the chance of being able to recover the original signal is to add **parity check bits** to it before sending it. These are deterministic functions of the original signal, which specify if the sum of the input bits is odd or even. This provides a form of **redundancy**, so that if one bit is corrupted, we can still infer its value, assuming the other bits are not flipped. (This is reasonable since we assume the bits are corrupted independently at random, so it is less likely that multiple bits are flipped than just one bit.)

For example, suppose we have two original message bits, and we add one parity bit. This can be modeled using a directed graphical model as shown in Figure 5.11(a). This graph encodes the following joint probability distribution:

$$p(\mathbf{x}, \mathbf{y}) = p(x_1)p(x_2)p(x_3|x_1, x_2) \prod_{i=1}^3 p(y_i|x_i) \quad (5.164)$$

The priors $p(x_1)$ and $p(x_2)$ are uniform. The conditional term $p(x_3|x_1, x_2)$ is deterministic, and computes the parity of (x_1, x_2) . In particular, we have $p(x_3 = 1|x_1, x_2) = 1$ if the total number of 1s in the block $x_{1:2}$ is odd. The likelihood terms $p(y_i|x_i)$ represent a bit flipping noisy channel model, with noise level $\alpha = 0.2$.

Suppose we observe $\mathbf{y} = (1, 0, 0)$. We know that this cannot be what the sender sent, since this violates the parity constraint (if $x_1 = 1$ then we know $x_3 = 1$). Instead, the 3 posterior modes for \mathbf{x} are 000 (first bit was flipped), 110 (second bit was flipped), and 101 (third bit was flipped). The only other configuration with non-zero support in the posterior is 011, which corresponds to the much less likely hypothesis that three bits were flipped (see Figure 5.11(b)). All other hypotheses (001, 010, and 100) are inconsistent with the deterministic method used to create codewords. (See Section 9.3.3.2 for further discussion of this point.)

In practice, we use more complex coding schemes that are more efficient, in the sense that they add less redundant bits to the message, but still guarantee that errors can be corrected. For details, see Section 9.4.8.

5.6 The information bottleneck

In this section, we discuss discriminative models $p(\mathbf{y}|\mathbf{x})$ that use a *stochastic bottleneck* between the input \mathbf{x} and the output \mathbf{y} to prevent overfitting, and improve robustness and calibration.

5.6.1 Vanilla IB

We say that \mathbf{z} is a **representation** of \mathbf{x} if \mathbf{z} is a (possibly stochastic) function of \mathbf{x} , and hence can be described by the conditional $p(\mathbf{z}|\mathbf{x})$. We say that a representation \mathbf{z} of \mathbf{x} is **sufficient** for task \mathbf{y} if $\mathbf{y} \perp \mathbf{x} | \mathbf{z}$, or equivalently, if $\mathbb{I}(\mathbf{z}; \mathbf{y}) = \mathbb{I}(\mathbf{x}; \mathbf{y})$, i.e., $\mathbb{H}(\mathbf{y}|\mathbf{z}) = \mathbb{H}(\mathbf{y}|\mathbf{x})$. We say that a representation is a **minimal sufficient statistic** if \mathbf{z} is sufficient and there is no other \mathbf{z} with smaller $\mathbb{I}(\mathbf{z}; \mathbf{x})$ value. Thus we would like to find a representation \mathbf{z} that maximizes $\mathbb{I}(\mathbf{z}; \mathbf{y})$ while minimizing $\mathbb{I}(\mathbf{z}; \mathbf{x})$. That is, we would like to optimize the following objective:

$$\min \beta \mathbb{I}(\mathbf{z}; \mathbf{x}) - \mathbb{I}(\mathbf{z}; \mathbf{y}) \quad (5.165)$$

where $\beta \geq 0$, and we optimize wrt the distributions $p(\mathbf{z}|\mathbf{x})$ and $p(\mathbf{y}|\mathbf{z})$. This is called the **information bottleneck principle** [TPB99]. This generalizes the concept of minimal sufficient statistic to take into account that there is a tradeoff between sufficiency and minimality, which is captured by the Lagrange multiplier $\beta > 0$.

This principle is illustrated in Figure 5.12. We assume Z is a function of X , but is independent of Y , i.e., we assume the graphical model $Z \leftarrow X \leftrightarrow Y$. This corresponds to the following joint distribution:

$$p(\mathbf{x}, \mathbf{y}, \mathbf{z}) = p(\mathbf{z}|\mathbf{x})p(\mathbf{y}|\mathbf{z})p(\mathbf{x}) \quad (5.166)$$

Thus Z can capture any amount of information about X that it wants, but cannot contain information that is unique to Y , as illustrated in Figure 5.12a. The optimal representation only captures information about X that is useful for Y ; to prevent us “wasting capacity” and fitting irrelevant details of the input, Z should also minimize information about X , as shown in Figure 5.12b.

If all the random variables are discrete, and $\mathbf{z} = e(\mathbf{x})$ is a deterministic function of \mathbf{x} , then the algorithm of [TPB99] can be used to minimize the IB objective in Section 5.6. The objective can also be solved analytically if all variables are jointly Gaussian [Che+05] (the resulting method can be viewed as a form of supervised PCA). But in general, it is intractable to solve this problem exactly. We discuss a tractable approximation in Section 5.6.2. (More details can be found in e.g., [SZ22].)

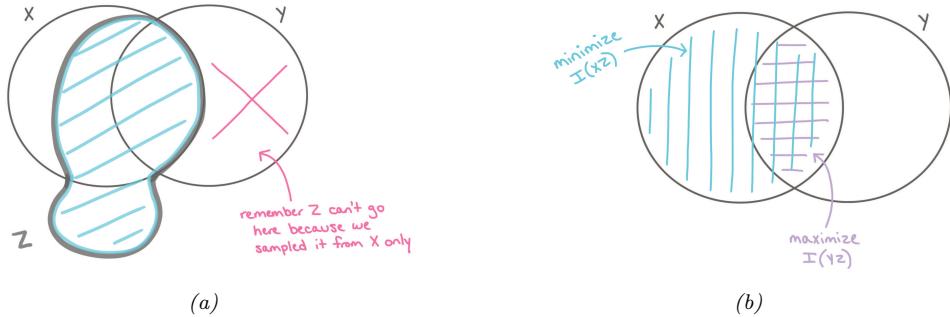


Figure 5.12: Information diagrams for information bottleneck. (a) Z can contain any amount of information about X (whether it useful for predicting Y or not), but it cannot contain information about Y that is not shared with X . (b) The optimal representation for Z maximizes $\mathbb{I}(Z, Y)$ and minimizes $\mathbb{I}(Z, X)$. Used with kind permission of Katie Everett.

5.6.2 Variational IB

In this section, we derive a variational upper bound on Equation (5.165), leveraging ideas from Section 5.3.6. This is called the **variational IB** or **VIB** method [Ale+16]. The key trick will be to use the non-negativity of the KL divergence to write

$$\int d\mathbf{x} p(\mathbf{x}) \log p(\mathbf{x}) \geq \int d\mathbf{x} p(\mathbf{x}) \log q(\mathbf{x}) \quad (5.167)$$

for any distribution q . (Note that both p and q may be conditioned on other variables.)

To explain the method in more detail, let us define the following notation. Let $e(\mathbf{z}|\mathbf{x}) = p(\mathbf{z}|\mathbf{x})$ represent the encoder, $b(\mathbf{z}|\mathbf{y}) \approx p(\mathbf{z}|\mathbf{y})$ represent the backwards encoder, $d(\mathbf{y}|\mathbf{z}) \approx p(\mathbf{y}|\mathbf{z})$ represent the classifier (decoder), and $m(\mathbf{z}) \approx p(\mathbf{z})$ represent the marginal. (Note that we get to choose $p(\mathbf{z}|\mathbf{x})$, but the other distributions are derived by approximations of the corresponding marginals and conditionals of the exact joint $p(\mathbf{x}, \mathbf{y}, \mathbf{z})$.) Also, let $\langle \cdot \rangle$ represent expectations wrt the relevant terms from the $p(\mathbf{x}, \mathbf{y}, \mathbf{z})$ joint.

With this notation, we can derive a lower bound on $\mathbb{I}(\mathbf{z}; \mathbf{y})$ as follows:

$$\mathbb{I}(\mathbf{z}; \mathbf{y}) = \int dy dz p(\mathbf{y}, \mathbf{z}) \log \frac{p(\mathbf{y}, \mathbf{z})}{p(\mathbf{y})p(\mathbf{z})} \quad (5.168)$$

$$= \int dy dz p(\mathbf{y}, \mathbf{z}) \log p(\mathbf{y}|\mathbf{z}) - \int dy dz p(\mathbf{y}, \mathbf{z}) \log p(\mathbf{y}) \quad (5.169)$$

$$= \int dy dz p(\mathbf{z})p(\mathbf{y}|\mathbf{z}) \log p(\mathbf{y}|\mathbf{z}) - \text{const} \quad (5.170)$$

$$\geq \int dy dz p(\mathbf{y}, \mathbf{z}) \log d(\mathbf{y}|\mathbf{z}) \quad (5.171)$$

$$= \langle \log d(\mathbf{y}|\mathbf{z}) \rangle \quad (5.172)$$

where we exploited the fact that $\mathbb{H}(p(\mathbf{y}))$ is a constant that is independent of our representation.

Note that we can approximate the expectations by sampling from

$$p(\mathbf{y}, \mathbf{z}) = \int d\mathbf{x} p(\mathbf{x})p(\mathbf{y}|\mathbf{x})p(\mathbf{z}|\mathbf{x}) = \int d\mathbf{x} p(\mathbf{x}, \mathbf{y})e(\mathbf{z}|\mathbf{x}) \quad (5.173)$$

This is just the empirical distribution “pushed through” the encoder.

Similarly, we can derive an upper bound on $\mathbb{I}(\mathbf{z}; \mathbf{x})$ as follows:

$$\mathbb{I}(\mathbf{z}; \mathbf{x}) = \int dz dx p(\mathbf{x}, \mathbf{z}) \log \frac{p(\mathbf{z}, \mathbf{x})}{p(\mathbf{x})p(\mathbf{z})} \quad (5.174)$$

$$= \int dz dx p(\mathbf{x}, \mathbf{z}) \log p(\mathbf{z}|\mathbf{x}) - \int dz p(\mathbf{z}) \log p(\mathbf{z}) \quad (5.175)$$

$$\leq \int dz dx p(\mathbf{x}, \mathbf{z}) \log p(\mathbf{z}|\mathbf{x}) - \int dz p(\mathbf{z}) \log m(\mathbf{z}) \quad (5.176)$$

$$= \int dz dx p(\mathbf{x}, \mathbf{z}) \log \frac{e(\mathbf{z}|\mathbf{x})}{m(\mathbf{z})} \quad (5.177)$$

$$= \langle \log e(\mathbf{z}|\mathbf{x}) \rangle - \langle \log m(\mathbf{z}) \rangle \quad (5.178)$$

Note that we can approximate the expectations by sampling from $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x})p(\mathbf{z}|\mathbf{x})$.

Putting it altogether, we get the following upper bound on the IB objective:

$$\beta \mathbb{I}(\mathbf{x}; \mathbf{z}) - \mathbb{I}(\mathbf{z}; \mathbf{y}) \leq \beta (\langle \log e(\mathbf{z}|\mathbf{x}) \rangle - \langle \log m(\mathbf{z}) \rangle) - \langle \log d(\mathbf{y}|\mathbf{z}) \rangle \quad (5.179)$$

Thus the VIB objective is

$$\mathcal{L}_{\text{VIB}} = \beta (\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})e(\mathbf{z}|\mathbf{x})} [\log e(\mathbf{z}|\mathbf{x}) - \log m(\mathbf{z})] - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})e(\mathbf{z}|\mathbf{x})d(\mathbf{y}|\mathbf{z})} [\log d(\mathbf{y}|\mathbf{z})]) \quad (5.180)$$

$$= -\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})e(\mathbf{z}|\mathbf{x})d(\mathbf{y}|\mathbf{z})} [\log d(\mathbf{y}|\mathbf{z})] + \beta \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(e(\mathbf{z}|\mathbf{x}) \parallel m(\mathbf{z}))] \quad (5.181)$$

We can now take stochastic gradients of this objective and minimize it (wrt the parameters of the encoder, decoder, and marginal) using SGD. (We assume the distributions are reparameterizable, as discussed in Section 6.3.5.) For the encoder $e(\mathbf{z}|\mathbf{x})$, we often use a conditional Gaussian, and for the decoder $d(\mathbf{y}|\mathbf{z})$, we often use a softmax classifier. For the marginal, $m(\mathbf{z})$, we should use a flexible model, such as a mixture of Gaussians, since it needs to approximate the **aggregated posterior** $p(\mathbf{z}) = \int dz p(\mathbf{x})e(\mathbf{z}|\mathbf{x})$, which is a mixture of N Gaussians (assuming $p(\mathbf{x})$ is an empirical distribution with N samples, and $e(\mathbf{z}|\mathbf{x})$ is a Gaussian).

We illustrate this in Figure 5.13, where we fit the an MLP model to MNIST. We use a 2d bottleneck layer before passing to the softmax. In panel a, we show the embedding learned by a deterministic encoder. We see that each image gets mapped to a point, and there is little overlap between classes, or between instances. In panels b-c, we show the embedding learned by a stochastic encoder. Each image gets mapped to a Gaussian distribution, we show the mean and the covariance separately. The classes are still well separated, but individual instances of a class are no longer distinguishable, since such information is not relevant for prediction purposes.

5.6.3 Conditional entropy bottleneck

The IB tries to maximize $\mathbb{I}(Z; Y)$ while minimizing $\mathbb{I}(Z; X)$. We can write this objective as

$$\min \mathbb{I}(\mathbf{x}; \mathbf{z}) - \lambda \mathbb{I}(\mathbf{y}; \mathbf{z}) \quad (5.182)$$

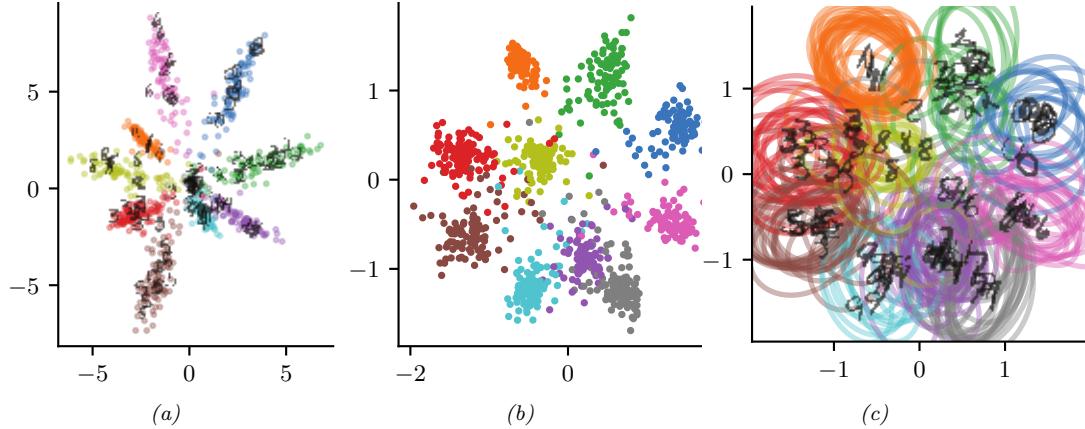


Figure 5.13: 2d embeddings of MNIST digits created by an MLP classifier. (a) Deterministic model. (b-c) VIB model, means and covariances. Generated by [vib_demo.ipynb](#). Used with kind permission of Alex Alemi.

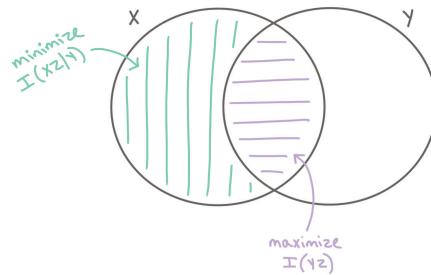


Figure 5.14: Conditional entropy bottleneck (CEB) chooses a representation Z that maximizes $\mathbb{I}(Z; Y)$ and minimizes $\mathbb{I}(X, Z|Y)$. Used with kind permission of Katie Everett.

for $\lambda \geq 0$. However, we see from the information diagram in Figure 5.12b that $\mathbb{I}(Z; X)$ contains some information that is relevant to Y . A sensible alternative objective is to minimizes the residual mutual information, $\mathbb{I}(X; Z|Y)$. This gives rise to the following objective:

$$\min \mathbb{I}(\mathbf{x}; \mathbf{z}|\mathbf{y}) - \lambda' \mathbb{I}(\mathbf{y}; \mathbf{z}) \quad (5.183)$$

for $\lambda' \geq 0$. This is known as the **conditional entropy bottleneck** or **CEB** [Fis20]. See Figure 5.14 for an illustration.

Since $\mathbb{I}(\mathbf{x}; \mathbf{z}|\mathbf{y}) = \mathbb{I}(\mathbf{x}; \mathbf{z}) - \mathbb{I}(\mathbf{y}; \mathbf{z})$, we see that the CEB is equivalent to standard IB with $\lambda' = \lambda + 1$. However, it is easier to upper bound $\mathbb{I}(\mathbf{x}; \mathbf{z}|\mathbf{y})$ than $\mathbb{I}(\mathbf{x}; \mathbf{z})$, since we are conditioning on \mathbf{y} , which

provides information about \mathbf{z} . In particular, by leveraging $p(\mathbf{z}|\mathbf{x}, \mathbf{y}) = p(\mathbf{z}|\mathbf{x})$ we have

$$\mathbb{I}(\mathbf{x}; \mathbf{z}|\mathbf{y}) = \mathbb{I}(\mathbf{x}; \mathbf{z}) - \mathbb{I}(\mathbf{y}; \mathbf{z}) \quad (5.184)$$

$$= \mathbb{H}(\mathbf{z}) - \mathbb{H}(\mathbf{z}|\mathbf{x}) - [\mathbb{H}(\mathbf{z}) - \mathbb{H}(\mathbf{z}|\mathbf{y})] \quad (5.185)$$

$$= -\mathbb{H}(\mathbf{z}|\mathbf{x}) + \mathbb{H}(\mathbf{z}|\mathbf{y}) \quad (5.186)$$

$$= \int dz d\mathbf{x} p(\mathbf{x}, \mathbf{z}) \log p(\mathbf{z}|\mathbf{x}) - \int dz d\mathbf{y} p(\mathbf{z}, \mathbf{y}) \log p(\mathbf{z}|\mathbf{y}) \quad (5.187)$$

$$\leq \int dz d\mathbf{x} p(\mathbf{x}, \mathbf{z}) \log e(\mathbf{z}|\mathbf{x}) - \int dz d\mathbf{y} p(\mathbf{z}, \mathbf{y}) \log b(\mathbf{z}|\mathbf{y}) \quad (5.188)$$

$$= \langle \log e(\mathbf{z}|\mathbf{x}) \rangle - \langle \log b(\mathbf{z}|\mathbf{y}) \rangle \quad (5.189)$$

Putting it altogether, we get the final CEB objective:

$$\min \beta (\langle \log e(\mathbf{z}|\mathbf{x}) \rangle - \langle \log b(\mathbf{z}|\mathbf{y}) \rangle) - \langle \log d(\mathbf{y}|\mathbf{z}) \rangle \quad (5.190)$$

Note that it is generally easier to learn the conditional backwards encoder $b(\mathbf{z}|\mathbf{y})$ than the unconditional marginal $m(\mathbf{z})$. Also, we know that the tightest upper bound occurs when $\mathbb{I}(\mathbf{x}; \mathbf{z}|\mathbf{y}) = \mathbb{I}(\mathbf{x}; \mathbf{z}) - \mathbb{I}(\mathbf{y}; \mathbf{z}) = 0$. The corresponding value of β corresponds to an optimal representation. By contrast, it is not clear how to measure distance from optimality when using IB.

6 Optimization

6.1 Introduction

In this chapter, we consider solving **optimization problems** of various forms. Abstractly these can all be written as

$$\boldsymbol{\theta}^* \in \operatorname{argmin}_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}) \quad (6.1)$$

where $\mathcal{L} : \Theta \rightarrow \mathbb{R}$ is the objective or loss function, and Θ is the parameter space we are optimizing over. However, this abstraction hides many details, such as whether the problem is constrained or unconstrained, discrete or continuous, convex or non-convex, etc. In the prequel to this book, [Mur22], we discussed some simple optimization algorithms for some common problems that arise in machine learning. In this chapter, we discuss some more advanced methods. For more details on optimization, please consult some of the many excellent textbooks, such as [KW19b; BV04; NW06; Ber15; Ber16] as well as various review articles, such as [BCN18; Sun+19b; PPS18; Pey20].

6.2 Automatic differentiation

This section is written by Roy Frostig.

This section is concerned with computing (partial) derivatives of complicated functions in an automatic manner. By “complicated” we mean those expressed as a composition of an arbitrary number of more basic operations, such as in deep neural networks. This task is known as **automatic differentiation (AD)**, or **autodiff**. AD is an essential component in optimization and deep learning, and is also used in several other fields across science and engineering. See e.g., Baydin et al. [Bay+15] for a review focused on machine learning and Griewank and Walther [GW08] for a classical textbook.

6.2.1 Differentiation in functional form

Before covering automatic differentiation, it is useful to review the mathematics of differentiation. We will use a particular **functional** notation for partial derivatives, rather than the typical one used throughout much of this book. We will refer to the latter as the **named variable** notation for the moment. Named variable notation relies on associating function arguments with names. For instance, given a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, the partial derivative of f with respect to its first scalar argument, at a

point $\mathbf{a} = (a_1, a_2)$, might be written:

$$\left. \frac{\partial f}{\partial x_1} \right|_{\mathbf{x}=\mathbf{a}} \quad (6.2)$$

This notation is not entirely self-contained. It refers to a name $\mathbf{x} = (x_1, x_2)$, implicit or inferred from context, suggesting the argument of f . An alternative expression is:

$$\frac{\partial}{\partial a_1} f(a_1, a_2) \quad (6.3)$$

where now a_1 serves both as an argument name (or a symbol in an expression) and as a particular evaluation point. Tracking names can become an increasingly complicated endeavor as we compose many functions together, each possibly taking several arguments.

A functional notation instead defines derivatives as operators on functions. If a function has multiple arguments, they are identified by position rather than by name, alleviating the need for auxiliary variable definitions. Some of the following definitions draw on those in Spivak's *Calculus on Manifolds* [Spi71] and in Sussman and Wisdom's *Functional Differential Geometry* [SW13], and generally appear more regularly in accounts of differential calculus and geometry. These texts are recommended for a more formal treatment, and a more mathematically general view, of the material briefly covered in this section.

Beside notation, we will rely on some basic multivariable calculus concepts. This includes the notion of (partial) derivatives, the differential or Jacobian of a function at a point, its role as a linear approximation local to the point, and various properties of linear maps, matrices, and transposition. We will focus on a finite-dimensional setting and write $\{\mathbf{e}_1, \dots, \mathbf{e}_n\}$ for the standard basis in \mathbb{R}^n .

Linear and multilinear functions. We use $F : \mathbb{R}^n \multimap \mathbb{R}^m$ to denote a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that is linear, and by $F[\mathbf{x}]$ its application to $\mathbf{x} \in \mathbb{R}^n$. Recall that such a linear map corresponds to a matrix in $\mathbb{R}^{m \times n}$ whose columns are $F[\mathbf{e}_1], \dots, F[\mathbf{e}_n]$; both interpretations will prove useful. Conveniently, function composition and matrix multiplication expressions look similar: to compose two linear maps F and G we can write $F \circ G$ or, barely abusing notation, consider the matrix FG . Every linear map $F : \mathbb{R}^n \multimap \mathbb{R}^m$ has a transpose $F : \mathbb{R}^m \multimap \mathbb{R}^n$, which is another linear map identified with transposing the corresponding matrix.

Repeatedly using the linear arrow symbol, we can denote by:

$$T : \underbrace{\mathbb{R}^n \multimap \cdots \multimap \mathbb{R}^n}_{k \text{ times}} \multimap \mathbb{R}^m \quad (6.4)$$

a multilinear, or more specifically k -linear, map:

$$T : \underbrace{\mathbb{R}^n \times \cdots \times \mathbb{R}^n}_{k \text{ times}} \rightarrow \mathbb{R}^m \quad (6.5)$$

which corresponds to an array (or tensor) in $\mathbb{R}^{m \times n \times \cdots \times n}$. We denote by $T[\mathbf{x}_1, \dots, \mathbf{x}_k] \in \mathbb{R}^m$ the application of such a k -linear map to vectors $\mathbf{x}_1, \dots, \mathbf{x}_k \in \mathbb{R}^n$.

The derivative operator. For an open set $U \subset \mathbb{R}^n$ and a differentiable function $f : U \rightarrow \mathbb{R}^m$, denote its **derivative function**:

$$\partial f : U \rightarrow (\mathbb{R}^n \multimap \mathbb{R}^m) \quad (6.6)$$

or equivalently $\partial f : U \rightarrow \mathbb{R}^{m \times n}$. This function maps a point $\mathbf{x} \in U$ to the Jacobian of all partial derivatives evaluated at \mathbf{x} . The symbol ∂ itself denotes the **derivative operator**, a function mapping functions to their derivative functions. When $m = 1$, the map $\partial f(\mathbf{x})$ recovers the standard gradient $\nabla f(\mathbf{x})$ at any $\mathbf{x} \in U$, by considering the matrix view of the former. Indeed, the nabla symbol ∇ is sometimes described as an operator as well, such that ∇f is a function. When $n = m = 1$, the Jacobian is scalar-valued, and ∂f is the familiar derivative f' .

In the expression $\partial f(\mathbf{x})[\mathbf{v}]$, we will sometimes refer to the argument \mathbf{x} as the **linearization point** for the Jacobian, and to \mathbf{v} as the **perturbation**. We call the map:

$$(\mathbf{x}, \mathbf{v}) \mapsto \partial f(\mathbf{x})[\mathbf{v}] \quad (6.7)$$

over linearization points $\mathbf{x} \in U$ and *input* perturbations $\mathbf{v} \in \mathbb{R}^n$ the **Jacobian-vector product (JVP)**. We similarly call its transpose:

$$(\mathbf{x}, \mathbf{u}) \mapsto \partial f(\mathbf{x})^\top[\mathbf{u}] \quad (6.8)$$

over linearization points $\mathbf{x} \in U$ and *output* perturbations $\mathbf{u} \in \mathbb{R}^m$ the **vector-Jacobian product (VJP)**.

Thinking about maps instead of matrices can help us define higher-order derivatives recursively, as we proceed to do below. It separately suggests how the action of a Jacobian is commonly written in code. When we consider writing $\partial f(\mathbf{x})$ in a program for a fixed \mathbf{x} , we often implement it as a function that carries out multiplication by the Jacobian matrix, i.e., $\mathbf{v} \mapsto \partial f(\mathbf{x})[\mathbf{v}]$, instead of explicitly representing it as a matrix of numbers in memory. Going a step further, for that matter, we often implement ∂f as an entire JVP at once, i.e., over any linearization point \mathbf{x} and perturbation \mathbf{v} . As a toy example with scalars, consider the cosine:

$$(x, v) \mapsto \partial \cos(x)v = -v \sin(x) \quad (6.9)$$

If we express this at once in code, we can, say, avoid computing $\sin(x)$ whenever $v = 0$.¹

Higher-order derivatives. Suppose the function f above remains arbitrarily differentiable over its domain $U \subset \mathbb{R}^n$. To take another derivative, we write:

$$\partial^2 f : U \rightarrow (\mathbb{R}^n \multimap \mathbb{R}^n \multimap \mathbb{R}^m) \quad (6.10)$$

where $\partial^2 f(\mathbf{x})$ is a bilinear map representing all second-order partial derivatives. In named variable notation, one might write $\frac{\partial f(\mathbf{x})}{\partial x_i \partial x_j}$ to refer to $\partial^2 f(\mathbf{x})[\mathbf{e}_i, \mathbf{e}_j]$, for example.

1. This example ignores that such an optimization might be done (best) by a compiler. Then again, for more complex examples, implementing $(\mathbf{x}, \mathbf{v}) \mapsto \partial f(\mathbf{x})[\mathbf{v}]$ as a single subroutine can help guide compiler optimizations all the same.

The second derivative function $\partial^2 f$ can be treated coherently as the outcome of applying the derivative operator twice. That is, it makes sense to say that $\partial^2 = \partial \circ \partial$. This observation extends recursively to cover arbitrary higher-order derivatives. For $k \geq 1$:

$$\partial^k f : U \rightarrow (\underbrace{\mathbb{R}^n \multimap \dots \multimap \mathbb{R}^n}_{k \text{ times}} \multimap \mathbb{R}^m) \quad (6.11)$$

is such that $\partial^k f(\mathbf{x})$ is a k -linear map.

With $m = 1$, the map $\partial^2 f(\mathbf{x})$ corresponds to the Hessian matrix at any $\mathbf{x} \in U$. Although Jacobians and Hessians suffice to make sense of many machine learning techniques, arbitrary higher-order derivatives are not hard to come by either (e.g., [Kel+20]). As an example, they appear when writing down something as basic as a function's Taylor series approximation, which we can express with our derivative operator as:

$$f(\mathbf{x} + \mathbf{v}) \approx f(\mathbf{x}) + \partial f(\mathbf{x})[\mathbf{v}] + \frac{1}{2!} \partial^2 f(\mathbf{x})[\mathbf{v}, \mathbf{v}] + \dots + \frac{1}{k!} \partial^k f(\mathbf{x})[\mathbf{v}, \dots, \mathbf{v}] \quad (6.12)$$

Multiple inputs. Now consider a function of two arguments:

$$g : U \times V \rightarrow \mathbb{R}^m. \quad (6.13)$$

where $U \subset \mathbb{R}^{n_1}$ and $V \subset \mathbb{R}^{n_2}$. For our purposes, a product domain like $U \times V$ mainly serves to suggest a convenient partitioning of a function's input components. It is isomorphic to a subset of $\mathbb{R}^{n_1+n_2}$, corresponding to a single-input function. The latter tells us how the derivative functions of g ought to look, based on previous definitions, and we will swap between the two views with little warning. Multiple inputs tend to arise in the context of computational circuits and programs: many functions in code are written to accept multiple arguments, and many basic operations (such as $+$) do the same.

With multiple inputs, we can denote by $\partial_i g$ the derivative function with respect to the i 'th argument:

$$\partial_1 g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \rightarrow (\mathbb{R}^{n_1} \multimap \mathbb{R}^m), \text{ and} \quad (6.14)$$

$$\partial_2 g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \rightarrow (\mathbb{R}^{n_2} \multimap \mathbb{R}^m). \quad (6.15)$$

Under the matrix view, the function $\partial_1 g$ maps a pair of points $\mathbf{x} \in \mathbb{R}^{n_1}$ and $\mathbf{y} \in \mathbb{R}^{n_2}$ to the matrix of all partial derivatives of g with respect to its first argument, evaluated at (\mathbf{x}, \mathbf{y}) . We take ∂g with no subscript to simply mean the concatenation of $\partial_1 g$ and $\partial_2 g$:

$$\partial g : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \rightarrow (\mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \multimap \mathbb{R}^m) \quad (6.16)$$

where, for every linearization point $(\mathbf{x}, \mathbf{y}) \in U \times V$ and perturbations $\dot{\mathbf{x}} \in \mathbb{R}^{n_1}$, $\dot{\mathbf{y}} \in \mathbb{R}^{n_2}$:

$$\partial g(\mathbf{x}, \mathbf{y})[\dot{\mathbf{x}}, \dot{\mathbf{y}}] = \partial_1 g(\mathbf{x}, \mathbf{y})[\dot{\mathbf{x}}] + \partial_2 g(\mathbf{x}, \mathbf{y})[\dot{\mathbf{y}}]. \quad (6.17)$$

Alternatively, taking the matrix view:

$$\partial g(\mathbf{x}, \mathbf{y}) = (\partial_1 g(\mathbf{x}, \mathbf{y}) \quad \partial_2 g(\mathbf{x}, \mathbf{y})). \quad (6.18)$$

This convention will simplify our chain rule statement below. When $n_1 = n_2 = m = 1$, both sub-matrices are scalar, and $\partial g_1(x, y)$ recovers the partial derivative that might otherwise be written in named variable notation as:

$$\frac{\partial}{\partial x} g(x, y). \quad (6.19)$$

However, the expression ∂g_1 bears a meaning on its own (as a function) whereas the expression $\frac{\partial g}{\partial x}$ may be ambiguous without further context. Again composing operators lets us write higher-order derivatives. For instance, $\partial_2 \partial_1 g(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^{m \times n_1 \times n_2}$, and if $m = 1$, the Hessian of g at (\mathbf{x}, \mathbf{y}) is:

$$\begin{pmatrix} \partial_1 \partial_1 g(\mathbf{x}, \mathbf{y}) & \partial_1 \partial_2 g(\mathbf{x}, \mathbf{y}) \\ \partial_2 \partial_1 g(\mathbf{x}, \mathbf{y}) & \partial_2 \partial_2 g(\mathbf{x}, \mathbf{y}) \end{pmatrix}. \quad (6.20)$$

Composition and fan-out. If $f = g \circ h$ for some $h : \mathbb{R}^n \rightarrow \mathbb{R}^p$ and $g : \mathbb{R}^p \rightarrow \mathbb{R}^m$, then the **chain rule** of calculus observes that:

$$\partial f(\mathbf{x}) = \partial g(h(\mathbf{x})) \circ \partial h(\mathbf{x}) \text{ for all } \mathbf{x} \in \mathbb{R}^n \quad (6.21)$$

How does this interact with our notation for multi-argument functions? For one, it can lead us to consider expressions with **fan-out**, where several sub-expressions are functions of the same input. For instance, assume two functions $a : \mathbb{R}^n \rightarrow \mathbb{R}^{m_1}$ and $b : \mathbb{R}^n \rightarrow \mathbb{R}^{m_2}$, and that:

$$f(\mathbf{x}) = g(a(\mathbf{x}), b(\mathbf{x})) \quad (6.22)$$

for some function g . Abbreviating $h(\mathbf{x}) = (a(\mathbf{x}), b(\mathbf{x}))$ so that $f(\mathbf{x}) = g(h(\mathbf{x}))$, Equations (6.16) and (6.21) tell us that:

$$\partial f(\mathbf{x}) = \partial g(h(\mathbf{x})) \circ \partial h(\mathbf{x}) \quad (6.23)$$

$$= \partial_1 g(a(\mathbf{x}), b(\mathbf{x})) \circ \partial a(\mathbf{x}) + \partial_2 g(a(\mathbf{x}), b(\mathbf{x})) \circ \partial b(\mathbf{x}) \quad (6.24)$$

Note that $+$ is meant pointwise here. It also follows from the above that if instead:

$$f(\mathbf{x}, \mathbf{y}) = g(a(\mathbf{x}), b(\mathbf{y})) \quad (6.25)$$

in other words, if we write multiple arguments but exhibit no fan-out, then:

$$\partial_1 f(\mathbf{x}, \mathbf{y}) = \partial_1 g(a(\mathbf{x}), b(\mathbf{y})) \circ \partial a(\mathbf{x}), \text{ and} \quad (6.26)$$

$$\partial_2 f(\mathbf{x}, \mathbf{y}) = \partial_2 g(a(\mathbf{x}), b(\mathbf{y})) \circ \partial b(\mathbf{y}) \quad (6.27)$$

Composition and fan-out rules for derivatives are what let us break down a complex derivative calculation into simpler ones. This is what automatic differentiation techniques rely on when processing the sort of elaborate numerical computations that turn up in modern machine learning and numerical programming.

6.2.2 Differentiating chains, circuits, and programs

The purpose of automatic differentiation is to compute derivatives of arbitrary functions provided as input. Given a function $f : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^m$ and a linearization point $\mathbf{x} \in U$, AD computes either:

- the JVP $\partial f(\mathbf{x})[\mathbf{v}]$ for an input perturbation $\mathbf{v} \in \mathbb{R}^n$, or
- the VJP $\partial f(\mathbf{x})^\top[\mathbf{u}]$ for an output perturbation $\mathbf{u} \in \mathbb{R}^m$.

In other words, JVPs and VJPs capture the two essential tasks of AD.²

Deciding what functions f to handle as input, and how to represent them, is perhaps the most load-bearing aspect of this setup. Over what *language* of functions should we operate? By a language, we mean some formal way of describing functions by composing a set of basic primitive operations. For primitives, we can think of various differentiable array operations (elementwise arithmetic, reductions, contractions, indexing and slicing, concatenation, etc.), but we will largely consider primitives and their derivatives as a given, and focus on how elaborately we can compose them. AD becomes increasingly challenging with increasingly expressive languages. Considering this, we introduce it in stages.

6.2.2.1 Chain compositions and the chain rule

To start, take only functions that are **chain compositions** of basic operations. Chains are a convenient class of function representations because derivatives *decompose* along the same structure according to the aptly-named chain rule.

As a toy example, consider $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ composed of three operations in sequence:

$$f = c \circ b \circ a \tag{6.28}$$

By the chain rule, its derivatives are given by

$$\partial f(\mathbf{x}) = \partial c(b(a(\mathbf{x}))) \circ \partial b(a(\mathbf{x})) \circ \partial a(\mathbf{x}) \tag{6.29}$$

Now consider the JVP against an input perturbation $\mathbf{v} \in \mathbb{R}^n$:

$$\partial f(\mathbf{x})[\mathbf{v}] = \partial c(b(a(\mathbf{x}))) [\partial b(a(\mathbf{x})) [\partial a(\mathbf{x})[\mathbf{v}]]] \tag{6.30}$$

This expression's bracketing highlights a right-to-left evaluation order that corresponds to **forward-mode automatic differentiation**. Namely, to carry out this JVP, it makes sense to compute prefixes of the original chain:

$$\mathbf{x}, a(\mathbf{x}), b(a(\mathbf{x})) \tag{6.31}$$

alongside the partial JVPs, because each is then immediately used as a subsequent linearization point, respectively:

$$\underline{\partial a(\mathbf{x})}, \underline{\partial b(a(\mathbf{x}))}, \underline{\partial c(b(a(\mathbf{x})))} \tag{6.32}$$

Extending this idea to arbitrary chain compositions gives Algorithm 6.1.

2. Materializing the Jacobian as a numerical array, as is commonly required in an optimization context, is a special case of computing a JVP or VJP against the standard basis vectors in \mathbb{R}^n or \mathbb{R}^m respectively.

Algorithm 6.1: Forward-mode automatic differentiation (JVP) on chains

```

1 input:  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  as a chain composition  $f = f_T \circ \dots \circ f_1$ 
2 input: linearization point  $\mathbf{x} \in \mathbb{R}^n$  and input perturbation  $\mathbf{v} \in \mathbb{R}^n$ 
3  $\mathbf{x}_0, \mathbf{v}_0 := \mathbf{x}, \mathbf{v}$ 
4 for  $t := 1, \dots, T$  do
5    $\mathbf{x}_t := f_t(\mathbf{x}_{t-1})$ 
6    $\mathbf{v}_t := \partial f_t(\mathbf{x}_{t-1})[\mathbf{v}_{t-1}]$ 
7 output:  $\mathbf{x}_T$ , equal to  $f(\mathbf{x})$ 
8 output:  $\mathbf{v}_T$ , equal to  $\partial f(\mathbf{x})[\mathbf{v}]$ 

```

By contrast, we can transpose Equation (6.29) to consider a VJP against an output perturbation $\mathbf{u} \in \mathbb{R}^m$:

$$\partial f(\mathbf{x})^\top [\mathbf{u}] = \partial a(\mathbf{x})^\top [\partial b(a(\mathbf{x}))^\top [\partial c(b(a(\mathbf{x})))^\top [\mathbf{u}]]] \quad (6.33)$$

Transposition reverses the Jacobian maps relative to their order in Equation (6.29), and now the bracketed evaluation corresponds to **reverse-mode automatic differentiation**. To carry out this VJP, we can compute the original chain prefixes \mathbf{x} , $a(\mathbf{x})$, and $b(a(\mathbf{x}))$ first, and then read them *in reverse* as successive linearization points:

$$\partial c(b(a(\mathbf{x})))^\top, \partial b(a(\mathbf{x}))^\top, \partial a(\mathbf{x})^\top \quad (6.34)$$

Extending this idea to arbitrary chain compositions gives Algorithm 6.2.

Algorithm 6.2: Reverse-mode automatic differentiation (VJP) on chains

```

1 input:  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  as a chain composition  $f = f_T \circ \dots \circ f_1$ 
2 input: linearization point  $\mathbf{x} \in \mathbb{R}^n$  and output perturbation  $\mathbf{u} \in \mathbb{R}^m$ 
3  $\mathbf{x}_0 := \mathbf{x}$ 
4 for  $t := 1, \dots, T$  do
5    $\mathbf{x}_t := f_t(\mathbf{x}_{t-1})$ 
6    $\mathbf{u}_T := \mathbf{u}$ 
7   for  $t := T, \dots, 1$  do
8      $\mathbf{u}_{t-1} := \partial f_t(\mathbf{x}_{t-1})^\top [\mathbf{u}_t]$ 
9   output:  $\mathbf{x}_T$ , equal to  $f(\mathbf{x})$ 
10  output:  $\mathbf{u}_0$ , equal to  $\partial f(\mathbf{x})^\top [\mathbf{u}]$ 

```

Although chain compositions impose a very specific structure, they already capture some deep neural network models, such as multi-layer perceptrons (provided matrix multiplication is a primitive operation), as covered in this book's prequel [Mur22, Ch.13].

Reverse-mode AD is faster than forward-mode when the output is scalar valued (as often arises in deep learning, where the output is a loss function). However, reverse-mode AD stores all chain

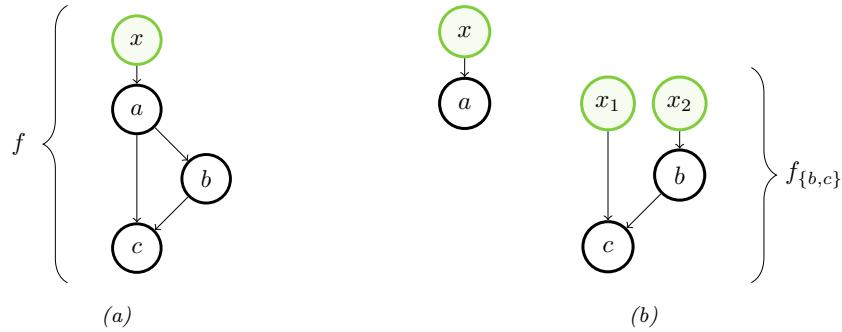


Figure 6.1: A circuit for a function f over three primitives, and its decomposition into two circuits without fan-out. Input nodes are drawn in green.

prefixes before its backwards traversal, so it consumes more memory than forward-mode. There are ways to combat this memory requirement in special-case scenarios, such as when the chained operations are each reversible [MDA15; Gom+17; KKL20]. One can also trade off memory for computation by discarding some prefixes and re-computing them as needed.

6.2.2.2 From chains to circuits

When primitives can accept multiple inputs, we can naturally extend chains to **circuits** — directed acyclic graphs over primitive operations, sometimes also called computation graphs. To set up for this section, we will distinguish between (1) **input nodes** of a circuit, which symbolize a function’s arguments, and (2) **primitive nodes**, each of which is labeled by a primitive operation. We assume that input nodes have no incoming edges and (without loss of generality) exactly one outgoing edge each, and that the graph has exactly one sink node. The overall function of the circuit is composition of operations from the input nodes to the sink, where the output of each operation is input to others according to its outgoing edges.

What made AD work in Section 6.2.2.1 is the fact that derivatives decompose along chains thanks to the aptly-named chain rule. When moving from chains to directed acyclic graphs, do we need some sort of “graph rule” in order to decompose our calculation along the circuit’s structure? Circuits introduce two new features: **fan-in** and **fan-out**. In graphical terms, fan-in simply refers to multiple edges incoming to a node, and fan-out refers to multiple edges outgoing.

What do these mean in functional terms? Fan-in happens when a primitive operation accepts multiple arguments. We observed in Section 6.2.1 that multiple arguments can be treated as one, and how the chain rule then applies. Fan-out requires slightly more care, specifically for reverse-mode differentiation.

The gist of an answer can be illustrated with a small example. Consider the circuit in Figure 6.1a. The operation a precedes b and c topologically, with an outgoing edge to each of both. We can cut a away from $\{b, c\}$ to produce two new circuits, shown in Figure 6.1b. The first corresponds to a and the second corresponds to the remaining computation, given by:

$$f_{\{b,c\}}(\mathbf{x}_1, \mathbf{x}_2) = c(\mathbf{x}_1, b(\mathbf{x}_2)). \quad (6.35)$$

We can recover the complete function f from a and $f_{\{b,c\}}$ with the help of a function dup given by:

$$\text{dup}(\mathbf{x}) = (\mathbf{x}, \mathbf{x}) \equiv \begin{pmatrix} I \\ I \end{pmatrix} \mathbf{x} \quad (6.36)$$

so that f can be written as a chain composition:

$$f = f_{\{b,c\}} \circ \text{dup} \circ a. \quad (6.37)$$

The circuit for $f_{\{b,c\}}$ contains no fan-out, and composition rules such as Equation (6.25) tell us its derivatives in terms of b , c , and their derivatives, all via the chain rule. Meanwhile, the chain rule applied to Equation (6.37) says that:

$$\partial f(\mathbf{x}) = \partial f_{\{b,c\}}(\text{dup}(a(\mathbf{x}))) \circ \partial \text{dup}(a(\mathbf{x})) \circ \partial a(\mathbf{x}) \quad (6.38)$$

$$= \partial f_{\{b,c\}}(a(\mathbf{x}), a(\mathbf{x})) \circ \begin{pmatrix} I \\ I \end{pmatrix} \circ \partial a(\mathbf{x}). \quad (6.39)$$

The above expression suggests calculating a JVP of f by right-to-left evaluation. It is similar to the JVP calculation suggested by Equation (6.30), but with a *duplication* operation $(I \ I)^T$ in the middle that arises from the Jacobian of dup .

Transposing the derivative of f at \mathbf{x} :

$$\partial f(\mathbf{x})^T = \partial a(\mathbf{x})^T \circ (I \ I) \circ \partial f_{\{b,c\}}(a(\mathbf{x}), a(\mathbf{x}))^T. \quad (6.40)$$

Considering right-to-left evaluation, this too is similar to the VJP calculation suggested by Equation (6.33), but with a *summation* operation $(I \ I)$ in the middle that arises from the *transposed* Jacobian of dup . The lesson of using dup in this small example is that, more generally, in order to handle fan-out in reverse mode AD, we can process operations in topological order — first forward and then in reverse — and then *sum* partial VJPs along multiple outgoing edges.

Algorithm 6.3: Foward-mode circuit differentiation (JVP)

- 1 **input:** $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ composing f_1, \dots, f_T in topological order, where f_1 is identity
 - 2 **input:** linearization point $\mathbf{x} \in \mathbb{R}^n$ and perturbation $\mathbf{v} \in \mathbb{R}^n$
 - 3 $\mathbf{x}_1, \mathbf{v}_1 := \mathbf{x}, \mathbf{v}$
 - 4 **for** $t := 2, \dots, T$ **do**
 - 5 let $[q_1, \dots, q_r] = \text{Pa}(t)$
 - 6 $\mathbf{x}_t := f_t(\mathbf{x}_{q_1}, \dots, \mathbf{x}_{q_r})$
 - 7 $\mathbf{v}_t := \sum_{i=1}^r \partial_i f_t(\mathbf{x}_{q_1}, \dots, \mathbf{x}_{q_r})[\mathbf{v}_{q_i}]$
 - 8 **output:** \mathbf{x}_T , equal to $f(\mathbf{x})$
 - 9 **output:** \mathbf{v}_T , equal to $\partial f(\mathbf{x})[\mathbf{v}]$
-

Algorithms 6.3 and 6.4 give a complete description of forward- and reverse-mode differentiation on circuits. For brevity they assume a single argument to the entire circuit function. Nodes are indexed $1, \dots, T$. The first is the input node, and the remaining $T - 1$ are labeled by their operation f_2, \dots, f_T . We take f_1 to be the identity. For each t , if f_t takes k arguments, let $\text{Pa}(t)$ be the ordered

Algorithm 6.4: Reverse-mode circuit differentiation (VJP)

```

1 input:  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  composing  $f_1, \dots, f_T$  in topological order, where  $f_1, f_T$  are identity
2 input: linearization point  $\mathbf{x} \in \mathbb{R}^n$  and perturbation  $\mathbf{u} \in \mathbb{R}^m$ 
3  $\mathbf{x}_1 := \mathbf{x}$ 
4 for  $t := 2, \dots, T$  do
5   let  $[q_1, \dots, q_r] = \text{Pa}(t)$ 
6    $\mathbf{x}_t := f_t(\mathbf{x}_{q_1}, \dots, \mathbf{x}_{q_r})$ 
7  $\mathbf{u}_{(T-1) \rightarrow T} := \mathbf{u}$ 
8 for  $t := T - 1, \dots, 2$  do
9   let  $[q_1, \dots, q_r] = \text{Pa}(t)$ 
10   $\mathbf{u}'_t := \sum_{c \in \text{Ch}(t)} \mathbf{u}_{t \rightarrow c}$ 
11   $\mathbf{u}_{q_i \rightarrow t} := \partial_i f_t(\mathbf{x}_{q_1}, \dots, \mathbf{x}_{q_r})^\top \mathbf{u}'_t$  for  $i = 1, \dots, r$ 
12 output:  $\mathbf{x}_T$ , equal to  $f(\mathbf{x})$ 
13 output:  $\mathbf{u}_{1 \rightarrow 2}$ , equal to  $\partial f(\mathbf{x})^\top \mathbf{u}$ 

```

list of k indices of its parent nodes (possibly containing duplicates, due to fan-out), and let $\text{Ch}(t)$ be the indices of its children (again possibly duplicate). Algorithm 6.4 takes a few more conventions: that f_T is the identity, that node T has $T - 1$ as its only parent, and that the child of node 1 is node 2.

Fan-out is a feature of *graphs*, but arguably not an essential feature of *functions*. One can always remove all fan-out from a circuit representation by duplicating nodes. Our interest in fan-out is precisely to avoid this, allowing for an efficient representation and, in turn, efficient memory use in Algorithms 6.3 and 6.4.

Reverse-mode AD on circuits has appeared under various names and formulations over the years. The algorithm is precisely the **backpropagation** algorithm in neural networks, a term introduced in the 1980s [RHW86b; RHW86a], and has separately come up in the context of control theory and sensitivity, as summarized in historical notes by Goodfellow, Bengio, and Courville [GBC16, Section 6.6].

6.2.2.3 From circuits to programs

Graphs are useful for introducing AD algorithms, and they might align well enough with neural network applications. But computer scientists have spent decades formalizing and studying various “languages for expressing functions compositionally”. Simply put, this is what programming languages are for! Can we automatically differentiate numerical functions expressed in, say, Python, Haskell, or some variant of the lambda calculus? These offer a far more widespread — and intuitively more expressive — way to describe an input function.³

In the previous sections, our approach to AD became more complex as we allowed for more complex graph structure. Something similar happens when we introduce grammatical constructs in a

³. In Python, what the language calls a “function” does not always describe a pure function of the arguments listed in its syntactic definition; its behavior may rely on side effects or global state, as allowed by the language. Here, we specifically mean a Python function that is pure and functional. JAX’s documentation details this restriction [Bra+18].

programming language. How do we adapt AD to handle a language with loops, conditionals, and recursive calls? What about parallel programming constructs? We have partial answers to questions like these today, although they invite a deeper dive into language details such as type systems and implementation concerns [Yu+18; Inn20; Pas+21b].

One example language construct that we already know how to handle, due to Section 6.2.2.2, is a standard `let` expression. In languages with a means of name or variable binding, multiple appearances of the same variable are analogous to fan-out in a circuit. Figure 6.1a corresponds to a function f that we could write in a functional language as:

```
f(x) =
  let ax = a(x)
    in c(ax, b(ax))
```

in which `ax` indeed appears twice after it is bound.

Understanding the interaction between language capacity and automatic differentiability is an ongoing topic of computer science research [PS08a; AP19; Vyt+19; BMP19; MP21]. In the meantime, functional languages have proven quite effective in recent AD systems, both widely-used and experimental. Systems such as JAX, Dex, and others are designed around pure functional programming models, and internally rely on functional program representations for differentiation [Mac+15; BPS16; Sha+19; FJL18; Bra+18; Mac+19; Dex; Fro+21; Pas+21a].

6.3 Stochastic optimization

In this section, we consider optimization of stochastic objectives of the form

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{z})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{z})] \quad (6.41)$$

where $\boldsymbol{\theta}$ are the parameters we are optimizing, and \boldsymbol{z} is a random variable, such as an external noise.

6.3.1 Stochastic gradient descent

Suppose we have a way of computing an unbiased estimate \mathbf{g}_t of the gradient of the objective function, i.e.,

$$\mathbb{E}[\mathbf{g}_t] = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t} \quad (6.42)$$

Then we can use this inside of a gradient descent procedure:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{g}_t \quad (6.43)$$

where η_t is the **learning rate** or **step size**. This is called **stochastic gradient descent** or **SGD**.

6.3.1.1 Choosing the step size

When using SGD, we need to be careful in how we choose the learning rate in order to achieve convergence. Rather than choosing a single constant learning rate, we can use a **learning rate**

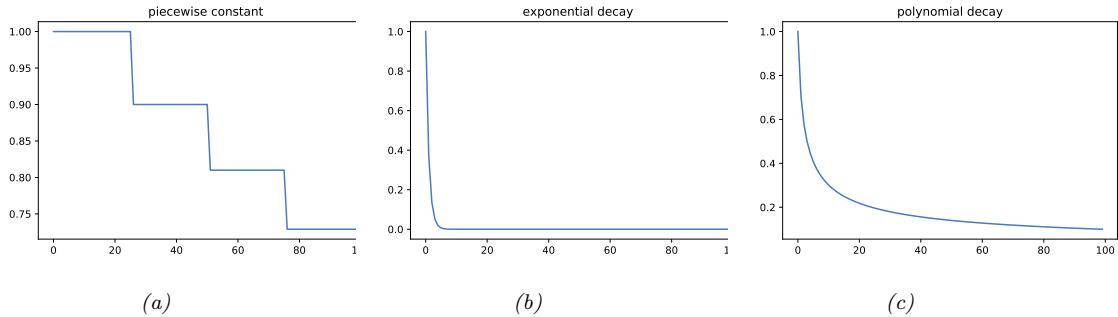


Figure 6.2: Illustration of some common learning rate schedules. (a) Piecewise constant. (b) Exponential decay. (c) Polynomial decay. Generated by [learning_rate_plot.ipynb](#).

schedule, in which we adjust the step size over time. Theoretically, a sufficient condition for SGD to achieve convergence is if the learning rate schedule satisfies the **Robbins-Monro conditions**:

$$\eta_t \rightarrow 0, \frac{\sum_{t=1}^{\infty} \eta_t^2}{\sum_{t=1}^{\infty} \eta_t} \rightarrow 0 \quad (6.44)$$

Some common examples of learning rate schedules are listed below:

$$\eta_t = \eta_i \text{ if } t_i \leq t \leq t_{i+1} \quad \text{piecewise constant} \quad (6.45)$$

$$\eta_t = \eta_0 e^{-\lambda t} \quad \text{exponential decay} \quad (6.46)$$

$$\eta_t = \eta_0 (\beta t + 1)^{-\alpha} \quad \text{polynomial decay} \quad (6.47)$$

In the piecewise constant schedule, t_i are a set of time points at which we adjust the learning rate to a specified value. For example, we may set $\eta_i = \eta_0 \gamma^i$, which reduces the initial learning rate by a factor of γ for each threshold (or milestone) that we pass. Figure 6.2a illustrates this for $\eta_0 = 1$ and $\gamma = 0.9$. This is called **step decay**. Sometimes the threshold times are computed adaptively, by estimating when the train or validation loss has plateaued; this is called **reduce-on-plateau**. Exponential decay is typically too fast, as illustrated in Figure 6.2b. A common choice is polynomial decay, with $\alpha = 0.5$ and $\beta = 1$, as illustrated in Figure 6.2c; this corresponds to a **square-root schedule**, $\eta_t = \eta_0 \frac{1}{\sqrt{t+1}}$. For more details, see [Mur22, Sec 8.4.3].

6.3.1.2 Variance reduction

SGD can be slow to converge because it relies on a stochastic estimate of the gradient. Various methods have been proposed for reducing the variance of the parameter estimates generated at each step, which can speedup convergence. For more details, see [Mur22, Sec 8.4.5].

6.3.1.3 Preconditioned SGD

In many cases, the gradient magnitudes can be very different along each dimension, corresponding to the loss surface being steep along some directions and shallow along others, similar to a valley

floor. In such cases, one can get faster convergence by scaling the gradient vector by a **conditioning matrix** \mathbf{C}_t as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{C}_t \mathbf{g}_t \quad (6.48)$$

This is called **preconditioned SGD**. For more details, see [Mur22, Sec 8.4.6].

6.3.2 SGD for optimizing a finite-sum objective

In the simplest case, the distribution used to compute the expectation, $q_{\boldsymbol{\theta}}(\mathbf{z})$, does not depend on the parameters being optimized, $\boldsymbol{\theta}$. In this case, we can push gradients inside the expectation operator, and then use Monte Carlo sampling for \mathbf{z} to approximate the gradient:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] = \mathbb{E}_{q(\mathbf{z})} [\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] \approx \frac{1}{S} \sum_{s=1}^S \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}_s) \quad (6.49)$$

For example, consider the problem of **empirical risk minimization** or **ERM**, which requires minimizing

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}_n) = \frac{1}{N} \sum_{n=1}^N \ell(\mathbf{y}_n, f(\mathbf{x}_n; \boldsymbol{\theta})) \quad (6.50)$$

where $\mathbf{z}_n = (\mathbf{x}_n, \mathbf{y}_n)$ is the n 'th labeled example, and f is a prediction function. This kind of objective is called a **finite sum objective**. We can write this as an expected loss wrt the empirical distribution $p_{\mathcal{D}}(\mathbf{x}, \mathbf{y})$:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{z})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] \quad (6.51)$$

Since the expectation depends on the data, and not on the parameters, we can approximate the gradient by using a **minibatch** of $B = |\mathcal{B}|$ datapoints from the full dataset \mathcal{D} at each iteration:

$$\mathbf{g}_t = \nabla \mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{B} \sum_{n \in \mathcal{B}} \nabla \ell(\mathbf{y}_n, f(\mathbf{x}_n; \boldsymbol{\theta})) \quad (6.52)$$

These noisy gradients can then be passed to SGD. When the dataset is large, this method is much faster than **full batch** gradient descent, since it does not require evaluating the loss on all N examples before updating the model [BB08; BB11].

6.3.3 SGD for optimizing the parameters of a distribution

Now suppose the stochasticity depends on the parameters we are optimizing. For example, \mathbf{z} could be an action sampled from a stochastic policy $q_{\boldsymbol{\theta}}$, as in RL (Section 35.3.2), or \mathbf{z} could be a latent variable sampled from an inference network $q_{\boldsymbol{\theta}}$, as in stochastic variational inference (see Section 10.2). In this case, the gradient is given by

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\theta}}(\mathbf{z})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] = \nabla_{\boldsymbol{\theta}} \int \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) q_{\boldsymbol{\theta}}(\mathbf{z}) d\mathbf{z} = \int \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) q_{\boldsymbol{\theta}}(\mathbf{z}) d\mathbf{z} \quad (6.53)$$

$$= \int [\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] q_{\boldsymbol{\theta}}(\mathbf{z}) d\mathbf{z} + \int \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) [\nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(\mathbf{z})] d\mathbf{z} \quad (6.54)$$

In the first line, we have assumed that we can swap the order of integration and differentiation (see [Moh+20] for discussion). In the second line, we use the product rule for derivatives.

The first term can be approximated by Monte Carlo sampling:

$$\int \left[\nabla_{\theta} \tilde{\mathcal{L}}(\theta, z) \right] q_{\theta}(z) dz \approx \frac{1}{S} \sum_{s=1}^S \nabla_{\theta} \tilde{\mathcal{L}}(\theta, z_s) \quad (6.55)$$

where $z_s \sim q_{\theta}$. Note that if $\tilde{\mathcal{L}}()$ is independent of θ , this term vanishes.

Now consider the second term, that takes the gradients of the distribution itself:

$$I \triangleq \int \tilde{\mathcal{L}}(\theta, z) [\nabla_{\theta} q_{\theta}(z)] dz \quad (6.56)$$

We can no longer use vanilla Monte Carlo sampling to approximate this integral. However, there are various other ways to approximate this (see [Moh+20] for an extensive review). We briefly describe the two main methods in Section 6.3.4 and Section 6.3.5.

6.3.4 Score function estimator (REINFORCE)

The simplest way to approximate Equation (6.56) is to exploit the **log derivative trick**, which is the following identity:

$$\nabla_{\theta} q_{\theta}(z) = q_{\theta}(z) \nabla_{\theta} \log q_{\theta}(z) \quad (6.57)$$

With this, we can rewrite Equation (6.56) as follows:

$$I = \int \tilde{\mathcal{L}}(\theta, z) [q_{\theta}(z) \nabla_{\theta} \log q_{\theta}(z)] dz = \mathbb{E}_{q_{\theta}(z)} \left[\tilde{\mathcal{L}}(\theta, z) \nabla_{\theta} \log q_{\theta}(z) \right] \quad (6.58)$$

This is called the **score function estimator** or **SFE** [Fu15]. (The term “score function” refers to the gradient of a log probability distribution, as explained in Section 3.3.4.1.) It is also called the **likelihood ratio gradient estimator**, or the **REINFORCE** estimator (the reason for this latter name is explained in Section 35.3.2). We can now easily approximate this with Monte Carlo:

$$I \approx \frac{1}{S} \sum_{s=1}^S \tilde{\mathcal{L}}(\theta, z_s) \nabla_{\theta} \log q_{\theta}(z_s) \quad (6.59)$$

where $z_s \sim q_{\theta}$. We only require that the sampling distribution is differentiable, not the objective $\tilde{\mathcal{L}}(\theta, z)$ itself. This allows the method to be used for blackbox stochastic optimization problems, such as variational optimization (Supplementary Section 6.4.3), black-box variational inference (Section 10.2.3), reinforcement learning (Section 35.3.2), etc.

6.3.4.1 Control variates

The score function estimate can have high variance. One way to reduce this is to use **control variates**, in which we replace $\tilde{\mathcal{L}}(\theta, z)$ with

$$\hat{\tilde{\mathcal{L}}}(\theta, z) = \tilde{\mathcal{L}}(\theta, z) - c(b(\theta, z) - \mathbb{E}[b(\theta, z)]) \quad (6.60)$$

where $b(\boldsymbol{\theta}, \mathbf{z})$ is a **baseline function** that is correlated with $\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})$, and $c > 0$ is a coefficient. Since $\mathbb{E}[\hat{\tilde{\mathcal{L}}}(\boldsymbol{\theta}, \mathbf{z})] = \mathbb{E}[\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})]$, we can use $\hat{\tilde{\mathcal{L}}}$ to compute unbiased gradient estimates of $\tilde{\mathcal{L}}$. The advantage is that this new estimate can result in lower variance, as we show in Section 11.6.3.

6.3.4.2 Rao-Blackwellization

Suppose $q_{\boldsymbol{\theta}}(\mathbf{z})$ is a discrete distribution. In this case, our objective becomes $\mathcal{L}(\boldsymbol{\theta}) = \sum_{\mathbf{z}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) q_{\boldsymbol{\theta}}(\mathbf{z})$. We can now easily compute gradients using $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{\mathbf{z}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) \nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(\mathbf{z})$. Of course, if \mathbf{z} can take on exponentially many values (e.g., we are optimizing over the space of strings), this expression is intractable. However, suppose we can partition this sum into two sets, a small set S_1 of high probability values and a large set S_2 of all other values. Then we can enumerate over S_1 and use the score function estimator for S_2 :

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \sum_{\mathbf{z} \in S_1} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) \nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(\mathbf{z}) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\mathbf{z} | \mathbf{z} \in S_2)} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) \nabla_{\boldsymbol{\theta}} \log q_{\boldsymbol{\theta}}(\mathbf{z})] \quad (6.61)$$

To compute the second expectation, we can use rejection sampling applied to samples from $q_{\boldsymbol{\theta}}(\mathbf{z})$. This procedure is a form of Rao-Blackwellization as shown in [Liu+19b], and reduces the variance compared to standard SFE (see Section 11.6.2 for details on Rao-Blackwellization).

6.3.5 Reparameterization trick

The score function estimator can have high variance, even when using a control variate. In this section, we derive a lower variance estimator, which can be applied if $\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})$ is differentiable wrt \mathbf{z} . We additionally require that we can compute a sample from $q_{\boldsymbol{\theta}}(\mathbf{z})$ by first sampling $\boldsymbol{\epsilon}$ from some noise distribution q_0 which is independent of $\boldsymbol{\theta}$, and then transforming to \mathbf{z} using a deterministic and differentiable function $\mathbf{z} = g(\boldsymbol{\theta}, \boldsymbol{\epsilon})$. For example, instead of sampling $\mathbf{z} \sim \mathcal{N}(\mu, \sigma^2)$, we can sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$ and compute

$$\mathbf{z} = g(\boldsymbol{\theta}, \boldsymbol{\epsilon}) = \mu + \sigma \boldsymbol{\epsilon} \quad (6.62)$$

where $\boldsymbol{\theta} = (\mu, \sigma)$. This allows us to rewrite our stochastic objective as follows:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\mathbf{z})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})] = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [\tilde{\mathcal{L}}(\boldsymbol{\theta}, g(\boldsymbol{\theta}, \boldsymbol{\epsilon}))] \quad (6.63)$$

Since $q_0(\boldsymbol{\epsilon})$ is independent of $\boldsymbol{\theta}$, we can push the gradient operator inside the expectation, which we can approximate with Monte Carlo:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{q_0(\boldsymbol{\epsilon})} [\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, g(\boldsymbol{\theta}, \boldsymbol{\epsilon}))] \approx \frac{1}{S} \sum_{s=1}^S \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, g(\boldsymbol{\theta}, \boldsymbol{\epsilon}_s)) \quad (6.64)$$

where $\boldsymbol{\epsilon}_s \sim q_0$. This is called the **reparameterization gradient** or the **pathwise derivative** [Gla03; Fu15; KW14; RMW14a; TLG14; JO18; FMM18], and is widely used in variational inference (Section 10.2.1). For a review of such methods, see [Moh+20].

Note that the tensorflow probability library (which also has a JAX interface) supports reparameterizable distributions. Therefore you can just write code in a straightforward way, as shown in ?? 6.1.

Listing 6.1: Derivative of a stochastic function

```
def expected_loss(params):
    zs = dist.sample(N, key)
    return jnp.mean(loss(params, zs))
g = jax.grad(expected_loss)(params)
```

6.3.5.1 Example

As a simple example, suppose we define some arbitrary function, such as $\tilde{\mathcal{L}}(z) = z^2 - 3z$, and then define its expected value as $\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\mathcal{N}(z|\mu,v)} [\tilde{\mathcal{L}}(z)]$, where $\boldsymbol{\theta} = (\mu, v)$ and $v = \sigma^2$. Suppose we want to compute

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \left[\frac{\partial}{\partial \mu} \mathbb{E} [\tilde{\mathcal{L}}(z)], \frac{\partial}{\partial v} \mathbb{E} [\tilde{\mathcal{L}}(z)] \right] \quad (6.65)$$

Since the Gaussian distribution is reparameterizable, we can sample $z \sim \mathcal{N}(z|\mu, v)$, and then use automatic differentiation to compute each of these gradient terms, and then average.

However, in the special case of Gaussian distributions, we can also compute the gradient vector directly. In particular, in Section 6.4.5.1 we present Bonnet's theorem, which states that

$$\frac{\partial}{\partial \mu} \mathbb{E} [\tilde{\mathcal{L}}(z)] = \mathbb{E} \left[\frac{\partial}{\partial z} \tilde{\mathcal{L}}(z) \right] \quad (6.66)$$

Similarly, Price's theorem states that

$$\frac{\partial}{\partial v} \mathbb{E} [\tilde{\mathcal{L}}(z)] = 0.5 \mathbb{E} \left[\frac{\partial^2}{\partial z^2} \tilde{\mathcal{L}}(z) \right] \quad (6.67)$$

In [gradient_expected_value_gaussian.ipynb](#) we show that these two methods are numerically equivalent, as theory suggests.

6.3.5.2 Total derivative

To compute the gradient term inside the expectation in Equation (6.64) we need to use the **total derivative**, since the function $\tilde{\mathcal{L}}$ depends on $\boldsymbol{\theta}$ directly and via the noise sample \mathbf{z} . Recall that, for a function of the form $\tilde{\mathcal{L}}(\theta_1, \dots, \theta_{d_\psi}, z_1(\boldsymbol{\theta}), \dots, z_{d_z}(\boldsymbol{\theta}))$, the total derivative wrt θ_i is given by the chain rule as follows:

$$\frac{\partial \tilde{\mathcal{L}}}{\partial \theta_i}^{\text{TD}} = \frac{\partial \tilde{\mathcal{L}}}{\partial \theta_i} + \sum_j \frac{\partial \tilde{\mathcal{L}}}{\partial z_j} \frac{\partial z_j}{\partial \theta_i} \quad (6.68)$$

and hence

$$\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z})^{\text{TD}} = \nabla_{\mathbf{z}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) \mathbf{J} + \nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) \quad (6.69)$$

where $\mathbf{J} = \frac{\partial \mathbf{z}^\top}{\partial \boldsymbol{\theta}}$ is the $d_z \times d_\psi$ Jacobian matrix of the noise transformation:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial z_1}{\partial \theta_1} & \cdots & \frac{\partial z_1}{\partial \theta_{d_\psi}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_{d_z}}{\partial \theta_1} & \cdots & \frac{\partial z_{d_z}}{\partial \theta_{d_\psi}} \end{pmatrix} \quad (6.70)$$

We leverage this decomposition in Section 6.3.5.3, where we derive a lower variance gradient estimator in the special case of variational inference.

6.3.5.3 “Sticking the landing” estimator

In this section we consider the special case which arises in variational inference (Section 10.2). The ELBO objective (for a single latent sample \mathbf{z}) has the form

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) = \log p(\mathbf{z}, \mathbf{x}) - \log q(\mathbf{z}|\boldsymbol{\theta}) \quad (6.71)$$

where $\boldsymbol{\theta}$ are the parameters of the variational posterior. The gradient becomes

$$\nabla_{\boldsymbol{\theta}} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathbf{z}) = \nabla_{\boldsymbol{\theta}} [\log p(\mathbf{z}, \mathbf{x}) - \log q(\mathbf{z}|\boldsymbol{\theta})] \quad (6.72)$$

$$= \underbrace{\nabla_{\mathbf{z}} [\log p(\mathbf{z}, \mathbf{x}) - \log q(\mathbf{z}|\boldsymbol{\theta})]}_{\text{path derivative}} \mathbf{J} - \underbrace{\nabla_{\boldsymbol{\theta}} \log q(\mathbf{z}|\boldsymbol{\theta})}_{\text{score function}} \quad (6.73)$$

The first term is the indirect effect of $\boldsymbol{\theta}$ on the objective via the generated samples \mathbf{z} . The second term is the direct effect of $\boldsymbol{\theta}$ on the objective. The second term is zero in expectation since it is the score function (see Equation (3.44)), but it may be non-zero for a finite number of samples, even if $q(\mathbf{z}|\boldsymbol{\theta}) = p(\mathbf{z}|\mathbf{x})$ is the true posterior. In [RWD17], they propose to drop the second term to create a lower variance estimator. This can be achieved by using $\log q(\mathbf{z}|\boldsymbol{\theta}')$, where $\boldsymbol{\theta}'$ is a “disconnected” copy of $\boldsymbol{\theta}$ that does not affect the gradient. In pseudocode, this looks like the following:

$$\epsilon \sim q_0(\epsilon) \quad (6.74)$$

$$\mathbf{z} = g(\epsilon, \boldsymbol{\theta}) \quad (6.75)$$

$$\boldsymbol{\theta}' = \text{stop-gradient}(\boldsymbol{\theta}) \quad (6.76)$$

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} [\log p(\mathbf{z}, \mathbf{x}) - \log q(\mathbf{z}|\boldsymbol{\theta}')] \quad (6.77)$$

They call this the **sticking the landing** or **STL** estimator.⁴ Note that the STL estimator is not always better than the “standard” estimator, without the stop gradient term. In [GD20], they propose to use a weighted combination of estimators, where the weights are optimized so as to reduce variance for a fixed amount of compute.

6.3.6 Gumbel softmax trick

When working with discrete variables, we cannot use the reparameterization trick. However, we can often relax the discrete variables to continuous ones in a way which allows the trick to be used, as we explain below.

Consider a one-hot vector \mathbf{d} with K bits, so $d_k \in \{0, 1\}$ and $\sum_{k=1}^K d_k = 1$. This can be used to represent a K -ary categorical variable d . Let $P(d) = \text{Cat}(d|\boldsymbol{\pi})$, where $\pi_k = P(d_k = 1)$, so $0 \leq \pi_k \leq 1$. Alternatively we can parameterize the distribution in terms of $(\alpha_1, \dots, \alpha_K)$, where $\pi_k = \alpha_k / (\sum_{k'=1}^K \alpha_{k'})$. We will denote this by $d \sim \text{Cat}(d|\boldsymbol{\alpha})$.

4. The expression “to stick a landing” means to land firmly on one’s feet after performing a gymnastics move. In the current context, the analogy is this: if the variational posterior is optimal, then we want our objective to be 0, and not to “wobble” with Monte Carlo noise.

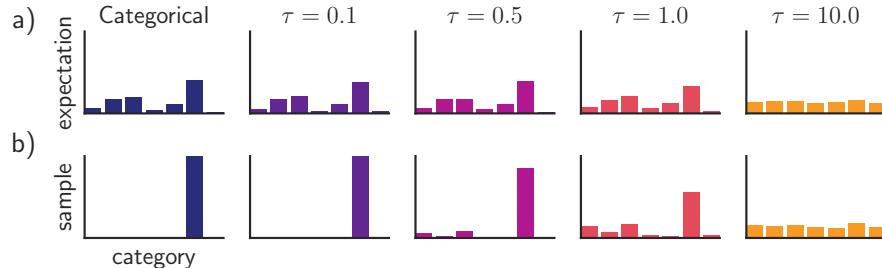


Figure 6.3: Illustration of the Gumbel-softmax (concrete) distribution with $K = 7$ states at different temperatures τ . The top row shows $E[z]$, and the bottom row shows samples $z \sim \text{GumbelSoftmax}(\alpha, \tau)$. The left column shows a discrete (categorical) distribution, which always produces one-hot samples. From Figure 1 of [JGP17]. Used with kind permission of Ben Poole.

We can sample a one-hot vector \mathbf{d} from this distribution by computing

$$\mathbf{d} = \text{onehot}(\underset{k}{\operatorname{argmax}} [\epsilon_k + \log \alpha_k]) \quad (6.78)$$

where $\epsilon_k \sim \text{Gumbel}(0, 1)$ is sampled from the **Gumbel distribution** [Gum54]. We can draw such samples by first sampling $u_k \sim \text{Unif}(0, 1)$ and then computing $\epsilon_k = -\log(-\log(u_k))$. This is called the **Gumbel-max trick** [MTM14], and gives us a reparameterizable representation for the categorical distribution.

Unfortunately, the derivative of the argmax is 0 everywhere except at the boundary of transitions from one label to another, where the derivative is undefined. However, suppose we replace the argmax with a softmax, and replace the discrete one-hot vector \mathbf{d} with a continuous relaxation $\mathbf{x} \in \Delta^{K-1}$, where $\Delta^{K-1} = \{\mathbf{x} \in \mathbb{R}^K : x_k \in [0, 1], \sum_{k=1}^K x_k = 1\}$ is the K -dimensional simplex. Then we can write

$$x_k = \frac{\exp((\log \alpha_k + \epsilon_k)/\tau)}{\sum_{k'=1}^K \exp((\log \alpha_{k'} + \epsilon_{k'})/\tau)} \quad (6.79)$$

where $\tau > 0$ is a temperature parameter. This is called the **Gumbel-softmax distribution** [JGP17] or the **concrete distribution** [MMT17]. This smoothly approaches the discrete distribution as $\tau \rightarrow 0$, as illustrated in Figure 6.3.

We can now replace $f(\mathbf{d})$ with $f(\mathbf{x})$, which allows us to take reparameterized gradients wrt \mathbf{x} .

6.3.7 Stochastic computation graphs

We can represent an arbitrary function containing both deterministic and stochastic components as a **stochastic computation graph**. We can then generalize the AD algorithm (Section 6.2) to leverage score function estimation (Section 6.3.4) and reparameterization (Section 6.3.5) to compute Monte Carlo gradients for complex nested functions. For details, see [Sch+15a; Gaj+19].

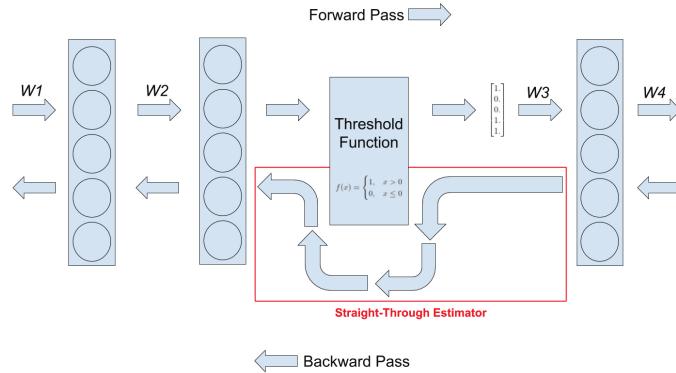


Figure 6.4: Illustration of straight-through estimator when applied to a binary threshold function in the middle of an MLP. From <https://www.hassanaskary.com/python/pytorch/deep%20learning/2020/09/19/intuitive-explanation-of-straight-through-estimators.html>. Used with kind permission of Hassan Askary.

6.3.8 Straight-through estimator

In this section, we discuss how to approximate the gradient of a quantized version of a signal. For example, suppose we have the following thresholding function, that binarizes its output:

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \quad (6.80)$$

This does not have a well-defined gradient. However, we can use the **straight-through estimator** proposed in [Ben13] as an approximation. The basic idea is to replace $g(x) = f'(x)$, where $f'(x)$ is the derivative of f wrt input, with $g(x) = x$ when computing the backwards pass. See Figure 6.4 for a visualization, and [Yin+19b] for an analysis of why this is a valid approximation.

In practice, we sometimes replace $g(x) = x$ with the **hard tanh** function, defined by

$$\text{HardTanh}(x) = \begin{cases} x & \text{if } -1 \leq x \leq 1 \\ 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \end{cases} \quad (6.81)$$

This ensures the gradients that are backpropagated don't get too large. See Section 21.6 for an application of this approach to discrete autoencoders.

6.4 Natural gradient descent

In this section, we discuss **natural gradient descent (NGD)** [Ama98], which is a second order method for optimizing the parameters of (conditional) probability distributions $p_{\theta}(\mathbf{y}|\mathbf{x})$. The key idea is to compute parameter updates by measuring distances between the induced distributions, rather than comparing parameter values directly.

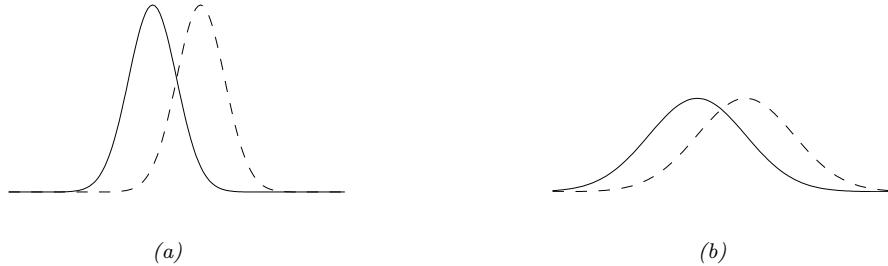


Figure 6.5: Changing the mean of a Gaussian by a fixed amount (from solid to dotted curve) can have more impact when the (shared) variance is small (as in a) compared to when the variance is large (as in b). Hence the impact (in terms of prediction accuracy) of a change to μ depends on where the optimizer is in (μ, σ) space. From Figure 3 of [Hon+10], reproduced from [Val00]. Used with kind permission of Antti Honkela.

For example, consider comparing two Gaussians, $p_{\boldsymbol{\theta}} = p(y|\mu, \sigma)$ and $p_{\boldsymbol{\theta}'} = p(y|\mu', \sigma')$. The (squared) Euclidean distance between the parameter vectors decomposes as $\|\boldsymbol{\theta} - \boldsymbol{\theta}'\|^2 = (\mu - \mu')^2 + (\sigma - \sigma')^2$. However, the predictive distribution has the form $\exp(-\frac{1}{2\sigma^2}(y - \mu)^2)$, so changes in μ need to be measured relative to σ . This is illustrated in Figure 6.5(a-b), which shows two univariate Gaussian distributions (dotted and solid lines) whose means differ by δ . In Figure 6.5(a), they share the same small variance σ^2 , whereas in Figure 6.5(b), they share the same large variance. It is clear that the value of δ matters much more (in terms of the effect on the distribution) when the variance is small. Thus we see that the two parameters interact with each other, which the Euclidean distance cannot capture. This problem gets much worse when we consider more complex models, such as deep neural networks. By modeling such correlations, NGD can converge much faster than other gradient methods.

6.4.1 Defining the natural gradient

The key to NGD is to measure the notion of distance between two probability distributions in terms of the KL divergence. As we show in Section 5.1.9, this can be approximated in terms of the Fisher information matrix (FIM). In particular, for any given input \mathbf{x} , we have

$$D_{\text{KL}}(p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}) \parallel p_{\boldsymbol{\theta}+\boldsymbol{\delta}}(\mathbf{y}|\mathbf{x})) \approx \frac{1}{2} \boldsymbol{\delta}^T \mathbf{F}_{\mathbf{x}} \boldsymbol{\delta} \quad (6.82)$$

where $\mathbf{F}_{\mathbf{x}}$ is the FIM

$$\mathbf{F}_{\mathbf{x}}(\boldsymbol{\theta}) = -\mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x})} [\nabla^2 \log p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x})] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x})} [(\nabla \log p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}))(\nabla \log p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}))^T] \quad (6.83)$$

We can compute the average KL between the current and updated distributions using $\frac{1}{2}\boldsymbol{\delta}^T \mathbf{F} \boldsymbol{\delta}$, where \mathbf{F} is the averaged FIM:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\mathbf{F}_{\mathbf{x}}(\boldsymbol{\theta})] \quad (6.84)$$

NGD uses the inverse FIM as a preconditioning matrix, i.e., we perform updates of the following form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{F}(\boldsymbol{\theta}_t)^{-1} \mathbf{g}_t \quad (6.85)$$

The term

$$\mathbf{F}^{-1} \mathbf{g}_t = \mathbf{F}^{-1} \nabla \mathcal{L}(\boldsymbol{\theta}_t) \triangleq \tilde{\nabla} \mathcal{L}(\boldsymbol{\theta}_t) \quad (6.86)$$

is called the **natural gradient**.

6.4.2 Interpretations of NGD

6.4.2.1 NGD as a trust region method

In [Supplementary](#) Section 6.1.3.1 we show that we can interpret standard gradient descent as optimizing a linear approximation to the objective subject to a penalty on the ℓ_2 norm of the change in parameters, i.e., if $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\delta}$, then we optimize

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \mathbf{g}_t^\top \boldsymbol{\delta} + \eta \|\boldsymbol{\delta}\|_2^2 \quad (6.87)$$

Now let us replace the squared distance with the squared FIM-based distance, $\|\boldsymbol{\delta}\|_F^2 = \boldsymbol{\delta}^\top \mathbf{F} \boldsymbol{\delta}$. This is equivalent to squared Euclidean distance in the **whitened coordinate system** $\boldsymbol{\phi} = \mathbf{F}^{\frac{1}{2}} \boldsymbol{\theta}$, since

$$\|\boldsymbol{\phi}_{t+1} - \boldsymbol{\phi}_t\|_2^2 = \|\mathbf{F}^{\frac{1}{2}}(\boldsymbol{\theta}_t + \boldsymbol{\delta}) - \mathbf{F}^{\frac{1}{2}} \boldsymbol{\theta}_t\|_2^2 = \|\mathbf{F}^{\frac{1}{2}} \boldsymbol{\delta}\|_2^2 = \|\boldsymbol{\delta}\|_F^2 \quad (6.88)$$

The new objective becomes

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \mathbf{g}_t^\top \boldsymbol{\delta} + \eta \boldsymbol{\delta}^\top \mathbf{F} \boldsymbol{\delta} \quad (6.89)$$

Solving $\nabla_{\boldsymbol{\delta}} M_t(\boldsymbol{\delta}) = \mathbf{0}$ gives the update

$$\boldsymbol{\delta}_t = -\eta \mathbf{F}^{-1} \mathbf{g}_t \quad (6.90)$$

This is the same as the natural gradient direction. Thus we can view NGD as a trust region method, where we use a first-order approximation to the objective, and use FIM-distance in the constraint.

In the above derivation, we assumed \mathbf{F} was a constant matrix. In most problems, it will change at each point in space, since we are optimizing in a curved space known as a **Riemannian manifold**. For certain models, we can compute the FIM efficiently, allowing us to capture curvature information, even though we use a first-order approximation to the objective.

6.4.2.2 NGD as a Gauss-Newton method

If $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$ is an exponential family distribution with natural parameters computed by $\boldsymbol{\eta} = f(\mathbf{x}, \boldsymbol{\theta})$, then one can show [Hes00; PB14] that NGD is identical to the generalized Gauss-Newton (GGN) method (Section 17.3.2). Furthermore, in the online setting, these methods are equivalent to performing sequential Bayesian inference using the extended Kalman filter, as shown in [Oll18].

6.4.3 Benefits of NGD

The use of the FIM as a preconditioning matrix, rather than the Hessian, has two advantages. First, \mathbf{F} is always positive definite, whereas \mathbf{H} can have negative eigenvalues at saddle points, which are prevalent in high dimensional spaces. Second, it is easy to approximate \mathbf{F} online from minibatches, since it is an expectation (wrt the empirical distribution) of outer products of gradient vectors. This is in contrast to Hessian-based methods [Byr+16; Liu+18a], which are much more sensitive to noise introduced by the minibatch approximation.

In addition, the connection with trust region optimization makes it clear that NGD updates parameters in a way that matter most for prediction, which allows the method to take larger steps in uninformative regions of parameter space, which can help avoid getting stuck on plateaus. This can also help with issues that arise when the parameters are highly correlated.

For example, consider a 2d Gaussian with an unusual, highly coupled parameterization, proposed in [SD12]:

$$p(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{2\pi} \exp \left[-\frac{1}{2} \left(x_1 - \left[3\theta_1 + \frac{1}{3}\theta_2 \right] \right)^2 - \frac{1}{2} \left(x_2 - \left[\frac{1}{3}\theta_1 \right] \right)^2 \right] \quad (6.91)$$

The objective is the cross entropy loss:

$$\mathcal{L}(\boldsymbol{\theta}) = -\mathbb{E}_{p^*(\mathbf{x})} [\log p(\mathbf{x}; \boldsymbol{\theta})] \quad (6.92)$$

The gradient of this objective is given by

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \begin{pmatrix} = \mathbb{E}_{p^*(\mathbf{x})} \left[3(x_1 - [3\theta_1 + \frac{1}{3}\theta_2]) + \frac{1}{3}(x_2 - [\frac{1}{3}\theta_1]) \right] \\ \mathbb{E}_{p^*(\mathbf{x})} \left[\frac{1}{3}(x_1 - [3\theta_1 + \frac{1}{3}\theta_2]) \right] \end{pmatrix} \quad (6.93)$$

Suppose that $p^*(\mathbf{x}) = p(\mathbf{x}; [0, 0])$. Then the Fisher matrix is a constant matrix, given by

$$\mathbf{F} = \begin{pmatrix} 3^2 + \frac{1}{3^2} & 1 \\ 1 & \frac{1}{3^2} \end{pmatrix} \quad (6.94)$$

Figure 6.6 compares steepest descent in $\boldsymbol{\theta}$ space with the natural gradient method, which is equivalent to steepest descent in ϕ space. Both methods start at $\boldsymbol{\theta} = (1, -1)$. The global optimum is at $\boldsymbol{\theta} = (0, 0)$. We see that the NG method (blue dots) converges much faster to this optimum and takes the shortest path, whereas steepest descent takes a very circuitous route. We also see that the gradient field in the whitened parameter space is more ‘‘spherical’’, which makes descent much simpler and faster.

Finally, note that since NGD is invariant to how we parameterize the distribution, we will get the same results even for a standard parameterization of the Gaussian. This is particularly useful if our probability model is more complex, such as a DNN (see e.g., [SSE18]).

6.4.4 Approximating the natural gradient

The main drawback of NGD is the computational cost of computing (the inverse of) the Fisher information matrix (FIM). To speed this up, several methods make assumptions about the form of \mathbf{F} , so it can be inverted efficiently. For example, [LeC+98] uses a diagonal approximation for

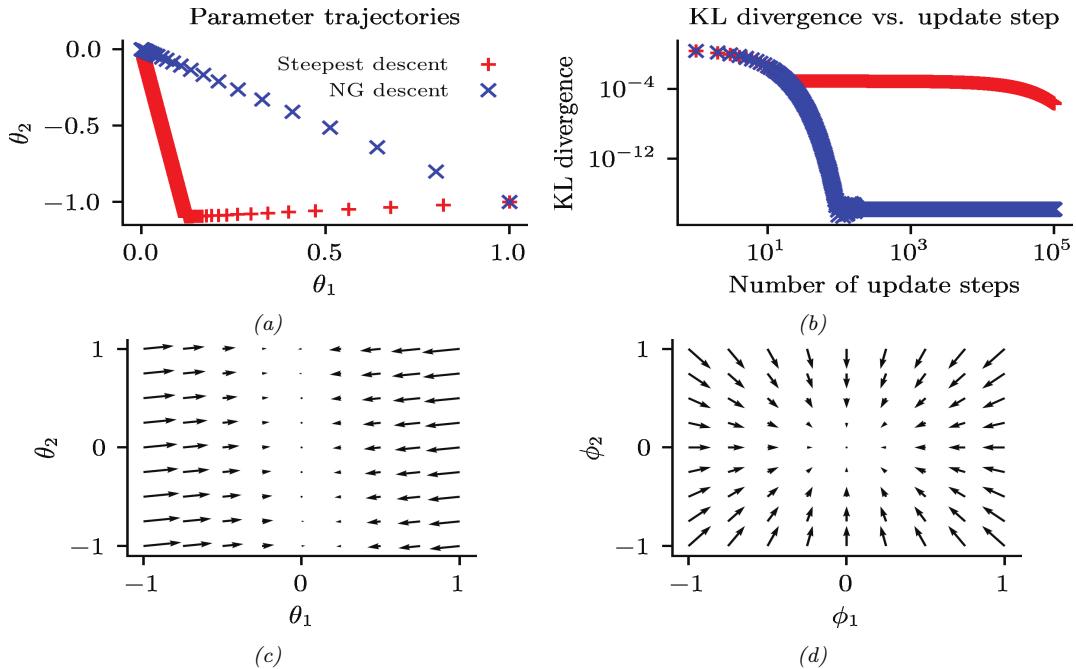


Figure 6.6: Illustration of the benefits of natural gradient vs steepest descent on a 2d problem. (a) Trajectories of the two methods in parameter space (red = steepest descent, blue = NG). They both start in the bottom right, at $(1, -1)$. (b) Objective vs number of iterations. (c) Gradient field in the θ parameter space. (d) Gradient field in the whitened $\phi = \mathbf{F}^{\frac{1}{2}}\theta$ parameter space used by NG. Generated by [nat_grad_demo.ipynb](#).

neural net training; [RMB08] uses a low-rank plus block diagonal approximation; and [GS15] assumes the covariance of the gradients can be modeled by a directed Gaussian graphical model with low treewidth (i.e., the Cholesky factorization of \mathbf{F} is sparse).

[MG15] propose the **KFAC** method, which stands for ‘‘Kronecker factored approximate curvature’’; this approximates the FIM of a DNN as a block diagonal matrix, where each block is a Kronecker product of two small matrices. This method has shown good results on supervised learning of neural nets [GM16; BGM17; Geo+18; Osa+19b] as well as reinforcement learning of neural policy networks [Wu+17]. The KFAC approximation can be justified using the mean field analysis of [AKO18]. In addition, [ZMG19] prove that KFAC will converge to the global optimum of a DNN if it is overparameterized (i.e., acts like an interpolator).

A simpler approach is to approximate the FIM by replacing the model’s distribution with the empirical distribution. In particular, define $p_{\mathcal{D}}(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{n=1}^N \delta_{\mathbf{x}_n}(\mathbf{x}) \delta_{\mathbf{y}_n}(\mathbf{y})$, $p_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^N \delta_{\mathbf{x}_n}(\mathbf{x})$

and $p_{\theta}(\mathbf{x}, \mathbf{y}) = p_{\mathcal{D}}(\mathbf{x})p(\mathbf{y}|\mathbf{x}, \theta)$. Then we can compute the **empirical Fisher** [Mar16] as follows:

$$\mathbf{F} = \mathbb{E}_{p_{\theta}(\mathbf{x}, \mathbf{y})} [\nabla \log p(\mathbf{y}|\mathbf{x}, \theta) \nabla \log p(\mathbf{y}|\mathbf{x}, \theta)^T] \quad (6.95)$$

$$\approx \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x}, \mathbf{y})} [\nabla \log p(\mathbf{y}|\mathbf{x}, \theta) \nabla \log p(\mathbf{y}|\mathbf{x}, \theta)^T] \quad (6.96)$$

$$= \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \nabla \log p(\mathbf{y}|\mathbf{x}, \theta) \nabla \log p(\mathbf{y}|\mathbf{x}, \theta)^T \quad (6.97)$$

This approximation is widely used, since it is simple to compute. In particular, we can compute a diagonal approximation using the squared gradient vector. (This is similar to ADAGRAD, but only uses the current gradient instead of a moving average of gradients; the latter is a better approach when performing stochastic optimization.)

Unfortunately, the empirical Fisher does not work as well as the true Fisher [KBH19; Tho+19]. To see why, note that when we reach a flat part of parameter space where the gradient vector goes to zero, the empirical Fisher will become singular, and hence the algorithm will get stuck on this plateau. However, the true Fisher takes expectations over the outputs, i.e., it marginalizes out \mathbf{y} . This will allow it to detect small changes in the output if we change the parameters. This is why the natural gradient method can “escape” plateaus better than standard gradient methods.

An alternative strategy is to use exact computation of \mathbf{F} , but solve for $\mathbf{F}^{-1}\mathbf{g}$ approximately using truncated conjugate gradient (CG) methods, where each CG step uses efficient methods for Hessian-vector products [Pea94]. This is called **Hessian free optimization** [Mar10a]. However, this approach can be slow, since it may take many CG iterations to compute a single parameter update.

6.4.5 Natural gradients for the exponential family

In this section, we assume \mathcal{L} is an expected loss of the following form:

$$\mathcal{L}(\boldsymbol{\mu}) = \mathbb{E}_{q_{\boldsymbol{\mu}}(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] \quad (6.98)$$

where $q_{\boldsymbol{\mu}}(\mathbf{z})$ is an exponential family distribution with moment parameters $\boldsymbol{\mu}$. This is the basis of variational optimization (discussed in [Supplementary](#) Section 6.4.3) and natural evolutionary strategies (discussed in [Section 6.7.6](#)).

It turns out the gradient wrt the moment parameters is the same as the natural gradient wrt the natural parameters $\boldsymbol{\lambda}$. This follows from the chain rule:

$$\frac{d}{d\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) = \frac{d\boldsymbol{\mu}}{d\boldsymbol{\lambda}} \frac{d}{d\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}) = \mathbf{F}(\boldsymbol{\lambda}) \nabla_{\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}) \quad (6.99)$$

where $\mathcal{L}(\boldsymbol{\mu}) = \mathcal{L}(\boldsymbol{\lambda}(\boldsymbol{\mu}))$, and where we used Equation (2.232) to write

$$\mathbf{F}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \boldsymbol{\mu}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}}^2 A(\boldsymbol{\lambda}) \quad (6.100)$$

Hence

$$\tilde{\nabla}_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) = \mathbf{F}(\boldsymbol{\lambda})^{-1} \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}) \quad (6.101)$$

It remains to compute the (regular) gradient wrt the moment parameters. The details on how to do this will depend on the form of the q and the form of $\mathcal{L}(\boldsymbol{\lambda})$. We discuss some approaches to this problem below.

6.4.5.1 Analytic computation for the Gaussian case

In this section, we assume that $q(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{m}, \mathbf{V})$. We now show how to compute the relevant gradients analytically.

Following Section 2.4.2.5, the natural parameters of q are

$$\boldsymbol{\lambda}^{(1)} = \mathbf{V}^{-1}\mathbf{m}, \quad \boldsymbol{\lambda}^{(2)} = -\frac{1}{2}\mathbf{V}^{-1} \quad (6.102)$$

and the moment parameters are

$$\boldsymbol{\mu}^{(1)} = \mathbf{m}, \quad \boldsymbol{\mu}^{(2)} = \mathbf{V} + \mathbf{m}\mathbf{m}^\top \quad (6.103)$$

For simplicity, we derive the result for the scalar case. Let $m = \mu^{(1)}$ and $v = \mu^{(2)} - (\mu^{(1)})^2$. By using the chain rule, the gradient wrt the moment parameters are

$$\frac{\partial \mathcal{L}}{\partial \mu^{(1)}} = \frac{\partial \mathcal{L}}{\partial m} \frac{\partial m}{\partial \mu^{(1)}} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial \mu^{(1)}} = \frac{\partial \mathcal{L}}{\partial m} - 2 \frac{\partial \mathcal{L}}{\partial v} m \quad (6.104)$$

$$\frac{\partial \mathcal{L}}{\partial \mu^{(2)}} = \frac{\partial \mathcal{L}}{\partial m} \frac{\partial m}{\partial \mu^{(2)}} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial \mu^{(2)}} = \frac{\partial \mathcal{L}}{\partial v} \quad (6.105)$$

It remains to compute the derivatives wrt m and v . If $\mathbf{z} \sim \mathcal{N}(\mathbf{m}, \mathbf{V})$, then from **Bonnet's theorem** [Bon64] we have

$$\frac{\partial}{\partial m_i} \mathbb{E} [\tilde{\mathcal{L}}(\mathbf{z})] = \mathbb{E} \left[\frac{\partial}{\partial \theta_i} \tilde{\mathcal{L}}(\mathbf{z}) \right] \quad (6.106)$$

And from **Price's theorem** [Pri58] we have

$$\frac{\partial}{\partial V_{ij}} \mathbb{E} [\tilde{\mathcal{L}}(\mathbf{z})] = c_{ij} \mathbb{E} \left[\frac{\partial^2}{\partial \theta_i \partial \theta_j} \tilde{\mathcal{L}}(\mathbf{z}) \right] \quad (6.107)$$

where $c_{ij} = \frac{1}{2}$ if $i = j$ and $c_{ij} = 1$ otherwise. (See `gradient_expected_value_gaussian.ipynb` for a “proof by example” of these claims.)

In the multivariate case, the result is as follows [OA09; KR21a]:

$$\nabla_{\boldsymbol{\mu}^{(1)}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] = \nabla_{\mathbf{m}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] - 2 \nabla_{\mathbf{V}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] \mathbf{m} \quad (6.108)$$

$$= \mathbb{E}_{q(\mathbf{z})} [\nabla_{\mathbf{z}} \tilde{\mathcal{L}}(\mathbf{z})] - \mathbb{E}_{q(\mathbf{z})} [\nabla_{\mathbf{z}}^2 \tilde{\mathcal{L}}(\mathbf{z})] \mathbf{m} \quad (6.109)$$

$$\nabla_{\boldsymbol{\mu}^{(2)}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] = \nabla_{\mathbf{V}} \mathbb{E}_{q(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] \quad (6.110)$$

$$= \frac{1}{2} \mathbb{E}_{q(\mathbf{z})} [\nabla_{\mathbf{z}}^2 \tilde{\mathcal{L}}(\mathbf{z})] \quad (6.111)$$

Thus we see that the natural gradients rely on both the gradient and Hessian of the loss function $\tilde{\mathcal{L}}(\mathbf{z})$. We will see applications of this result in [Supplementary](#) Section 6.4.2.2.

6.4.5.2 Stochastic approximation for the general case

In general, it can be hard to analytically compute the natural gradient. However, we can compute a Monte Carlo approximation. To see this, let us assume \mathcal{L} is an expected loss of the following form:

$$\mathcal{L}(\boldsymbol{\mu}) = \mathbb{E}_{q_{\boldsymbol{\mu}}(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] \quad (6.112)$$

From Equation (6.101) the natural gradient is given by

$$\nabla_{\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{\mu}) = \mathbf{F}(\boldsymbol{\lambda})^{-1} \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) \quad (6.113)$$

For exponential family distributions, both of these terms on the RHS can be written as expectations, and hence can be approximated by Monte Carlo, as noted by [KL17a]. To see this, note that

$$\mathbf{F}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \boldsymbol{\mu}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z})} [\mathcal{T}(\mathbf{z})] \quad (6.114)$$

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\lambda}) = \nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z})} [\tilde{\mathcal{L}}(\mathbf{z})] \quad (6.115)$$

If q is reparameterizable, we can apply the reparameterization trick (Section 6.3.5) to push the gradient inside the expectation operator. This lets us sample \mathbf{z} from q , compute the gradients, and average; we can then pass the resulting stochastic gradients to SGD.

6.4.5.3 Natural gradient of the entropy function

In this section, we discuss how to compute the natural gradient of the entropy of an exponential family distribution, which is useful when performing variational inference (Chapter 10). The natural gradient is given by

$$\tilde{\nabla}_{\boldsymbol{\lambda}} \mathbb{H}(\boldsymbol{\lambda}) = -\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\mathbf{z})} [\log q(\mathbf{z})] \quad (6.116)$$

where, from Equation (2.160), we have

$$\log q(\mathbf{z}) = \log h(\mathbf{z}) + \mathcal{T}(\mathbf{z})^T \boldsymbol{\lambda} - A(\boldsymbol{\lambda}) \quad (6.117)$$

Since $\mathbb{E}[\mathcal{T}(\mathbf{z})] = \boldsymbol{\mu}$, we have

$$\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\mathbf{z})} [\log q(\mathbf{z})] = \nabla_{\boldsymbol{\mu}} \mathbb{E}_{q(\mathbf{z})} [\log h(\mathbf{z})] + \nabla_{\boldsymbol{\mu}} \boldsymbol{\mu}^T \boldsymbol{\lambda}(\boldsymbol{\mu}) - \nabla_{\boldsymbol{\mu}} A(\boldsymbol{\lambda}) \quad (6.118)$$

where $h(\mathbf{z})$ is the base measure. Since $\boldsymbol{\lambda}$ is a function of $\boldsymbol{\mu}$, we have

$$\nabla_{\boldsymbol{\mu}} \boldsymbol{\mu}^T \boldsymbol{\lambda} = \boldsymbol{\lambda} + (\nabla_{\boldsymbol{\mu}} \boldsymbol{\lambda})^T \boldsymbol{\mu} = \boldsymbol{\lambda} + (\mathbf{F}_{\boldsymbol{\lambda}}^{-1} \nabla_{\boldsymbol{\lambda}} \boldsymbol{\lambda})^T \boldsymbol{\mu} = \boldsymbol{\lambda} + \mathbf{F}_{\boldsymbol{\lambda}}^{-1} \boldsymbol{\mu} \quad (6.119)$$

and since $\boldsymbol{\mu} = \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda})$ we have

$$\nabla_{\boldsymbol{\mu}} A(\boldsymbol{\lambda}) = \mathbf{F}_{\boldsymbol{\lambda}}^{-1} \nabla_{\boldsymbol{\lambda}} A(\boldsymbol{\lambda}) = \mathbf{F}_{\boldsymbol{\lambda}}^{-1} \boldsymbol{\mu} \quad (6.120)$$

Hence

$$-\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q_{\boldsymbol{\mu}}(\mathbf{z})} [\log q(\mathbf{z})] = -\nabla_{\boldsymbol{\mu}} \mathbb{E}_{q(\mathbf{z})} [\log h(\mathbf{z})] - \boldsymbol{\lambda} \quad (6.121)$$

If we assume that $h(\mathbf{z}) = \text{const}$, as is often the case, we get

$$\tilde{\nabla}_{\boldsymbol{\lambda}} \mathbb{H}(\boldsymbol{\lambda}) = -\boldsymbol{\lambda} \quad (6.122)$$

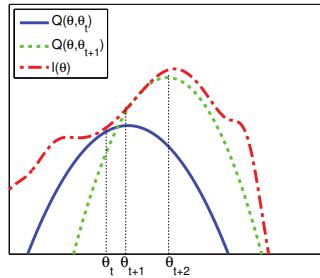


Figure 6.7: Illustration of a bound optimization algorithm. Adapted from Figure 9.14 of [Bis06]. Generated by [em_log_likelihood_max.ipynb](#).

6.5 Bound optimization (MM) algorithms

In this section, we consider a class of algorithms known as **bound optimization** or MM algorithms. In the context of minimization, MM stands for **majorize-minimize**. In the context of maximization, MM stands for **minorize-maximize**. There are many examples of MM algorithms, such as EM (Section 6.5.3), proximal gradient methods (Section 4.1), the mean shift algorithm for clustering [FH75; Che95; FT05], etc. For more details, see e.g., [HL04; Mai15; SBP17; Nad+19],

6.5.1 The general algorithm

In this section, we assume our goal is to *maximize* some function $\ell(\boldsymbol{\theta})$ wrt its parameters $\boldsymbol{\theta}$. The basic approach in MM algorithms is to construct a **surrogate function** $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t)$ which is a tight lowerbound to $\ell(\boldsymbol{\theta})$ such that $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \leq \ell(\boldsymbol{\theta})$ and $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$. If these conditions are met, we say that Q minorizes ℓ . We then perform the following update at each step:

$$\boldsymbol{\theta}^{t+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \quad (6.123)$$

This guarantees us monotonic increases in the original objective:

$$\ell(\boldsymbol{\theta}^{t+1}) \geq Q(\boldsymbol{\theta}^{t+1}, \boldsymbol{\theta}^t) \geq Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t) \quad (6.124)$$

where the first inequality follows since $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}')$ is a lower bound on $\ell(\boldsymbol{\theta}^t)$ for any $\boldsymbol{\theta}'$; the second inequality follows from Equation (6.123); and the final equality follows the tightness property. As a consequence of this result, if you do not observe monotonic increase of the objective, you must have an error in your math and/or code. This is a surprisingly powerful debugging tool.

This process is sketched in Figure 6.7. The dashed red curve is the original function (e.g., the log-likelihood of the observed data). The solid blue curve is the lower bound, evaluated at $\boldsymbol{\theta}^t$; this touches the objective function at $\boldsymbol{\theta}^t$. We then set $\boldsymbol{\theta}^{t+1}$ to the maximum of the lower bound (blue curve), and fit a new bound at that point (dotted green curve). The maximum of this new bound becomes $\boldsymbol{\theta}^{t+2}$, etc.

6.5.2 Example: logistic regression

If $\ell(\boldsymbol{\theta})$ is a concave function we want to maximize, then one way to obtain a valid lower bound is to use a bound on its Hessian, i.e., to find a negative definite matrix \mathbf{B} such that $\mathbf{H}(\boldsymbol{\theta}) \succ \mathbf{B}$. In this case, one can show (see [BCN18, App. B]) that

$$\ell(\boldsymbol{\theta}) \geq \ell(\boldsymbol{\theta}^t) + (\boldsymbol{\theta} - \boldsymbol{\theta}^t)^T \mathbf{g}(\boldsymbol{\theta}^t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^t)^T \mathbf{B}(\boldsymbol{\theta} - \boldsymbol{\theta}^t) \quad (6.125)$$

where $\mathbf{g}(\boldsymbol{\theta}^t) = \nabla \ell(\boldsymbol{\theta}^t)$. Therefore the following function is a valid lower bound:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \boldsymbol{\theta}^T (\mathbf{g}(\boldsymbol{\theta}^t) - \mathbf{B}\boldsymbol{\theta}^t) + \frac{1}{2}\boldsymbol{\theta}^T \mathbf{B}\boldsymbol{\theta} \quad (6.126)$$

The corresponding update becomes

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \mathbf{B}^{-1} \mathbf{g}(\boldsymbol{\theta}^t) \quad (6.127)$$

This is similar to a Newton update, except we use \mathbf{B} , which is a fixed matrix, rather than $\mathbf{H}(\boldsymbol{\theta}^t)$, which changes at each iteration. This can give us some of the advantages of second order methods at lower computational cost.

For example, let us fit a multi-class logistic regression model using MM. (We follow the presentation of [Kri+05], who also consider the more interesting case of *sparse* logistic regression.) The probability that example n belongs to class $c \in \{1, \dots, C\}$ is given by

$$p(y_n = c | \mathbf{x}_n, \mathbf{w}) = \frac{\exp(\mathbf{w}_c^T \mathbf{x}_n)}{\sum_{i=1}^C \exp(\mathbf{w}_i^T \mathbf{x}_n)} \quad (6.128)$$

Because of the normalization condition $\sum_{c=1}^C p(y_n = c | \mathbf{x}_n, \mathbf{w}) = 1$, we can set $\mathbf{w}_C = \mathbf{0}$. (For example, in binary logistic regression, where $C = 2$, we only learn a single weight vector.) Therefore the parameters $\boldsymbol{\theta}$ correspond to a weight matrix \mathbf{w} of size $D(C - 1)$, where $\mathbf{x}_n \in \mathbb{R}^D$.

If we let $\mathbf{p}_n(\mathbf{w}) = [p(y_n = 1 | \mathbf{x}_n, \mathbf{w}), \dots, p(y_n = C-1 | \mathbf{x}_n, \mathbf{w})]$ and $\mathbf{y}_n = [\mathbb{I}(y_n = 1), \dots, \mathbb{I}(y_n = C-1)]$, we can write the log-likelihood as follows:

$$\ell(\mathbf{w}) = \sum_{n=1}^N \left[\sum_{c=1}^{C-1} y_{nc} \mathbf{w}_c^T \mathbf{x}_n - \log \sum_{c=1}^C \exp(\mathbf{w}_c^T \mathbf{x}_n) \right] \quad (6.129)$$

The gradient is given by the following:

$$\mathbf{g}(\mathbf{w}) = \sum_{n=1}^N (\mathbf{y}_n - \mathbf{p}_n(\mathbf{w})) \otimes \mathbf{x}_n \quad (6.130)$$

where \otimes denotes Kronecker product (which, in this case, is just outer product of the two vectors). The Hessian is given by the following:

$$\mathbf{H}(\mathbf{w}) = - \sum_{n=1}^N (\text{diag}(\mathbf{p}_n(\mathbf{w})) - \mathbf{p}_n(\mathbf{w}) \mathbf{p}_n(\mathbf{w})^T) \otimes (\mathbf{x}_n \mathbf{x}_n^T) \quad (6.131)$$

We can construct a lower bound on the Hessian, as shown in [Boh92]:

$$\mathbf{H}(\mathbf{w}) \succ -\frac{1}{2}[\mathbf{I} - \mathbf{1}\mathbf{1}^\top/C] \otimes (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top) \triangleq \mathbf{B} \quad (6.132)$$

where \mathbf{I} is a $(C - 1)$ -dimensional identity matrix, and $\mathbf{1}$ is a $(C - 1)$ -dimensional vector of all 1s. In the binary case, this becomes

$$\mathbf{H}(\mathbf{w}) \succ -\frac{1}{2}(1 - \frac{1}{2})(\sum_{n=1}^N \mathbf{x}_n^\top \mathbf{x}_n) = -\frac{1}{4}\mathbf{X}^\top \mathbf{X} \quad (6.133)$$

This follows since $p_n \leq 0.5$ so $-(p_n - p_n^2) \geq -0.25$.

We can use this lower bound to construct an MM algorithm to find the MLE. The update becomes

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \mathbf{B}^{-1} \mathbf{g}(\mathbf{w}^t) \quad (6.134)$$

For example, let us consider the binary case, so $\mathbf{g}^t = \nabla \ell(\mathbf{w}^t) = \mathbf{X}^\top (\mathbf{y} - \boldsymbol{\mu}^t)$, where $\boldsymbol{\mu}^t = [p_n(\mathbf{w}^t), (1 - p_n(\mathbf{w}^t))]_{n=1}^N$. The update becomes

$$\mathbf{w}^{t+1} = \mathbf{w}^t - 4(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{g}^t \quad (6.135)$$

The above is faster (per step) than the IRLS (iteratively reweighted least squares) algorithm (i.e., Newton's method), which is the standard method for fitting GLMs. To see this, note that the Newton update has the form

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \mathbf{H}^{-1} \mathbf{g}(\mathbf{w}^t) = \mathbf{w}^t - (\mathbf{X}^\top \mathbf{S}^t \mathbf{X})^{-1} \mathbf{g}^t \quad (6.136)$$

where $\mathbf{S}^t = \text{diag}(\boldsymbol{\mu}^t \odot (1 - \boldsymbol{\mu}^t))$. We see that Equation (6.135) is faster to compute, since we can precompute the constant matrix $(\mathbf{X}^\top \mathbf{X})^{-1}$.

6.5.3 The EM algorithm

In this section, we discuss the **expectation maximization (EM)** algorithm [DLR77; MK07], which is an algorithm designed to compute the MLE or MAP parameter estimate for probability models that have **missing data** and/or **hidden variables**. It is a special case of an MM algorithm.

The basic idea behind EM is to alternate between estimating the hidden variables (or missing values) during the **E step** (expectation step), and then using the fully observed data to compute the MLE during the **M step** (maximization step). Of course, we need to iterate this process, since the expected values depend on the parameters, but the parameters depend on the expected values.

In Section 6.5.3.1, we show that EM is a **bound optimization** algorithm, which implies that this iterative procedure will converge to a local maximum of the log likelihood. The speed of convergence depends on the amount of missing data, which affects the tightness of the bound [XJ96; MD97; SRG03; KKS20].

We now describe the EM algorithm for a generic model. We let \mathbf{y}_n be the visible data for example n , and \mathbf{z}_n be the hidden data.

6.5.3.1 Lower bound

The goal of EM is to maximize the log likelihood of the observed data:

$$\ell(\boldsymbol{\theta}) = \sum_{n=1}^N \log p(\mathbf{y}_n | \boldsymbol{\theta}) = \sum_{n=1}^N \log \left[\sum_{\mathbf{z}_n} p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta}) \right] \quad (6.137)$$

where \mathbf{y}_n are the visible variables and \mathbf{z}_n are the hidden variables. Unfortunately this is hard to optimize, since the log cannot be pushed inside the sum.

EM gets around this problem as follows. First, consider a set of arbitrary distributions $q_n(\mathbf{z}_n)$ over each hidden variable \mathbf{z}_n . The observed data log likelihood can be written as follows:

$$\ell(\boldsymbol{\theta}) = \sum_{n=1}^N \log \left[\sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \frac{p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})}{q_n(\mathbf{z}_n)} \right] \quad (6.138)$$

Using Jensen's inequality, we can push the log (which is a concave function) inside the expectation to get the following lower bound on the log likelihood:

$$\ell(\boldsymbol{\theta}) \geq \sum_n \sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \log \frac{p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})}{q_n(\mathbf{z}_n)} \quad (6.139)$$

$$= \sum_n \underbrace{\mathbb{E}_{q_n} [\log p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})]}_{\mathcal{L}(\boldsymbol{\theta}, q_n | \mathbf{y}_n)} + \mathbb{H}(q_n) \quad (6.140)$$

$$= \sum_n \mathcal{L}(\boldsymbol{\theta}, q_n | \mathbf{y}_n) \triangleq \mathcal{L}(\boldsymbol{\theta}, \{q_n\} | \mathcal{D}) \quad (6.141)$$

where $\mathbb{H}(q)$ is the entropy of probability distribution q , and $\mathcal{L}(\boldsymbol{\theta}, \{q_n\} | \mathcal{D})$ is called the **evidence lower bound** or **ELBO**, since it is a lower bound on the log marginal likelihood, $\log p(\mathbf{y}_{1:N} | \boldsymbol{\theta})$, also called the evidence. Optimizing this bound is the basis of variational inference, as we discuss in Section 10.1.

6.5.3.2 E step

We see that the lower bound is a sum of N terms, each of which has the following form:

$$\mathcal{L}(\boldsymbol{\theta}, q_n | \mathbf{y}_n) = \sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \log \frac{p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})}{q_n(\mathbf{z}_n)} \quad (6.142)$$

$$= \sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \log \frac{p(\mathbf{z}_n | \mathbf{y}_n, \boldsymbol{\theta}) p(\mathbf{y}_n | \boldsymbol{\theta})}{q_n(\mathbf{z}_n)} \quad (6.143)$$

$$= \sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \log \frac{p(\mathbf{z}_n | \mathbf{y}_n, \boldsymbol{\theta})}{q_n(\mathbf{z}_n)} + \sum_{\mathbf{z}_n} q_n(\mathbf{z}_n) \log p(\mathbf{y}_n | \boldsymbol{\theta}) \quad (6.144)$$

$$= -D_{\text{KL}}(q_n(\mathbf{z}_n) \| p(\mathbf{z}_n | \mathbf{y}_n, \boldsymbol{\theta})) + \log p(\mathbf{y}_n | \boldsymbol{\theta}) \quad (6.145)$$

where $D_{\text{KL}}(q \| p) \triangleq \sum_z q(z) \log \frac{q(z)}{p(z)}$ is the Kullback-Leibler divergence (or KL divergence for short) between probability distributions q and p . We discuss this in more detail in Section 5.1, but the key

property we need here is that $D_{\text{KL}}(q \parallel p) \geq 0$ and $D_{\text{KL}}(q \parallel p) = 0$ iff $q = p$. Hence we can maximize the lower bound $\mathbb{L}(\boldsymbol{\theta}, \{q_n\} | \mathcal{D})$ wrt $\{q_n\}$ by setting each one to $q_n^* = p(\mathbf{z}_n | \mathbf{y}_n, \boldsymbol{\theta})$. This is called the **E step**. This ensures the ELBO is a tight lower bound:

$$\mathbb{L}(\boldsymbol{\theta}, \{q_n^*\} | \mathcal{D}) = \sum_n \log p(\mathbf{y}_n | \boldsymbol{\theta}) = \ell(\boldsymbol{\theta} | \mathcal{D}) \quad (6.146)$$

To see how this connects to bound optimization, let us define

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \mathbb{L}(\boldsymbol{\theta}, \{p(\mathbf{z}_n | \mathbf{y}_n; \boldsymbol{\theta}^t)\}) \quad (6.147)$$

Then we have $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \leq \ell(\boldsymbol{\theta})$ and $Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = \ell(\boldsymbol{\theta}^t)$, as required.

However, if we cannot compute the posteriors $p(\mathbf{z}_n | \mathbf{y}_n; \boldsymbol{\theta}^t)$ exactly, we can still use an approximate distribution $q(\mathbf{z}_n | \mathbf{y}_n; \boldsymbol{\theta}^t)$; this will yield a non-tight lower-bound on the log-likelihood. This generalized version of EM is known as variational EM [NH98b]. See Section 6.5.6.1 for details.

6.5.3.3 M step

In the M step, we need to maximize $\mathbb{L}(\boldsymbol{\theta}, \{q_n^t\})$ wrt $\boldsymbol{\theta}$, where the q_n^t are the distributions computed in the E step at iteration t . Since the entropy terms $\mathbb{H}(q_n)$ are constant wrt $\boldsymbol{\theta}$, we can drop them in the M step. We are left with

$$\ell^t(\boldsymbol{\theta}) = \sum_n \mathbb{E}_{q_n^t(\mathbf{z}_n)} [\log p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})] \quad (6.148)$$

This is called the **expected complete data log likelihood**. If the joint probability is in the exponential family (Section 2.4), we can rewrite this as

$$\ell^t(\boldsymbol{\theta}) = \sum_n \mathbb{E} [\mathcal{T}(\mathbf{y}_n, \mathbf{z}_n)^\top \boldsymbol{\theta} - A(\boldsymbol{\theta})] = \sum_n (\mathbb{E} [\mathcal{T}(\mathbf{y}_n, \mathbf{z}_n)]^\top \boldsymbol{\theta} - A(\boldsymbol{\theta})) \quad (6.149)$$

where $\mathbb{E} [\mathcal{T}(\mathbf{y}_n, \mathbf{z}_n)]$ are called the **expected sufficient statistics**.

In the M step, we maximize the expected complete data log likelihood to get

$$\boldsymbol{\theta}^{t+1} = \arg \max_{\boldsymbol{\theta}} \sum_n \mathbb{E}_{q_n^t} [\log p(\mathbf{y}_n, \mathbf{z}_n | \boldsymbol{\theta})] \quad (6.150)$$

In the case of the exponential family, the maximization can be solved in closed-form by matching the moments of the expected sufficient statistics (Section 2.4.5).

We see from the above that the E step does not in fact need to return the full set of posterior distributions $\{q(\mathbf{z}_n)\}$, but can instead just return the sum of the expected sufficient statistics, $\sum_n \mathbb{E}_{q(\mathbf{z}_n)} [\mathcal{T}(\mathbf{y}_n, \mathbf{z}_n)]$.

A common application of EM is for fitting mixture models; we discuss this in the prequel to this book, [Mur22]. Below we give a different example.

6.5.4 Example: EM for an MVN with missing data

It is easy to compute the MLE for a multivariate normal when we have a fully observed data matrix: we just compute the sample mean and covariance. In this section, we consider the case where we have

missing data or partially observed data. For example, we can think of the entries of \mathbf{Y} as being answers to a survey; some of these answers may be unknown. There are many kinds of missing data, as we discuss in Section 3.11. In this section, we make the missing at random (MAR) assumption, for simplicity. Under the MAR assumption, the log likelihood of the visible data has the form

$$\log p(\mathbf{X}|\boldsymbol{\theta}) = \sum_n \log p(\mathbf{x}_n|\boldsymbol{\theta}) = \sum_n \log \left[\int p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta}) d\mathbf{z}_n \right] \quad (6.151)$$

where \mathbf{x}_n are the visible variables in case n , \mathbf{z}_n are the hidden variables, and $\mathbf{y}_n = (\mathbf{z}_n, \mathbf{x}_n)$ are all the variables. Unfortunately, this objective is hard to maximize. since we cannot push the log inside the expectation. Fortunately, we can easily apply EM, as we explain below.

6.5.4.1 E step

Suppose we have the parameters $\boldsymbol{\theta}^{t-1}$ from the previous iteration. Then we can compute the expected complete data log likelihood at iteration t as follows:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{t-1}) = \mathbb{E} \left[\sum_{n=1}^N \log \mathcal{N}(\mathbf{y}_n|\boldsymbol{\mu}, \boldsymbol{\Sigma}) | \mathcal{D}, \boldsymbol{\theta}^{t-1} \right] \quad (6.152)$$

$$= -\frac{N}{2} \log |2\pi\boldsymbol{\Sigma}| - \frac{1}{2} \sum_n \mathbb{E} [(\mathbf{y}_n - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{y}_n - \boldsymbol{\mu})] \quad (6.153)$$

$$= -\frac{N}{2} \log |2\pi\boldsymbol{\Sigma}| - \frac{1}{2} \text{tr}(\boldsymbol{\Sigma}^{-1} \sum_n \mathbb{E} [(\mathbf{y}_n - \boldsymbol{\mu})(\mathbf{y}_n - \boldsymbol{\mu})^\top]) \quad (6.154)$$

$$= -\frac{N}{2} \log |\boldsymbol{\Sigma}| - \frac{ND}{2} \log(2\pi) - \frac{1}{2} \text{tr}(\boldsymbol{\Sigma}^{-1} \mathbb{E} [\mathbf{S}(\boldsymbol{\mu})]) \quad (6.155)$$

where

$$\mathbb{E} [\mathbf{S}(\boldsymbol{\mu})] \triangleq \sum_n \left(\mathbb{E} [\mathbf{y}_n \mathbf{y}_n^\top] + \boldsymbol{\mu} \boldsymbol{\mu}^\top - 2\boldsymbol{\mu} \mathbb{E} [\mathbf{y}_n]^\top \right) \quad (6.156)$$

(We drop the conditioning of the expectation on \mathcal{D} and $\boldsymbol{\theta}^{t-1}$ for brevity.) We see that we need to compute $\sum_n \mathbb{E} [\mathbf{y}_n]$ and $\sum_n \mathbb{E} [\mathbf{y}_n \mathbf{y}_n^\top]$; these are the expected sufficient statistics.

To compute these quantities, we use the results from Section 2.3.1.3. We have

$$p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_n|\mathbf{m}_n, \mathbf{V}_n) \quad (6.157)$$

$$\mathbf{m}_n \triangleq \boldsymbol{\mu}_h + \boldsymbol{\Sigma}_{hv} \boldsymbol{\Sigma}_{vv}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_v) \quad (6.158)$$

$$\mathbf{V}_n \triangleq \boldsymbol{\Sigma}_{hh} - \boldsymbol{\Sigma}_{hv} \boldsymbol{\Sigma}_{vv}^{-1} \boldsymbol{\Sigma}_{vh} \quad (6.159)$$

where we partition $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ into blocks based on the hidden and visible indices h and v . Hence the expected sufficient statistics are

$$\mathbb{E} [\mathbf{y}_n] = (\mathbb{E} [\mathbf{z}_n]; \mathbf{x}_n) = (\mathbf{m}_n; \mathbf{x}_n) \quad (6.160)$$

To compute $\mathbb{E}[\mathbf{y}_n \mathbf{y}_n^\top]$, we use the result that $\text{Cov}[\mathbf{y}] = \mathbb{E}[\mathbf{y}\mathbf{y}^\top] - \mathbb{E}[\mathbf{y}]\mathbb{E}[\mathbf{y}^\top]$. Hence

$$\mathbb{E}[\mathbf{y}_n \mathbf{y}_n^\top] = \mathbb{E}\left[\begin{pmatrix} \mathbf{z}_n \\ \mathbf{x}_n \end{pmatrix} (\mathbf{z}_n^\top \quad \mathbf{x}_n^\top)\right] = \begin{pmatrix} \mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] & \mathbb{E}[\mathbf{z}_n] \mathbf{x}_n^\top \\ \mathbf{x}_n \mathbb{E}[\mathbf{z}_n]^\top & \mathbf{x}_n \mathbf{x}_n^\top \end{pmatrix} \quad (6.161)$$

$$\mathbb{E}[\mathbf{z}_n \mathbf{z}_n^\top] = \mathbb{E}[\mathbf{z}_n] \mathbb{E}[\mathbf{z}_n]^\top + \mathbf{V}_n \quad (6.162)$$

6.5.4.2 M step

By solving $\nabla Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) = \mathbf{0}$, we can show that the M step is equivalent to plugging these ESS into the usual MLE equations to get

$$\boldsymbol{\mu}^t = \frac{1}{N} \sum_n \mathbb{E}[\mathbf{y}_n] \quad (6.163)$$

$$\boldsymbol{\Sigma}^t = \frac{1}{N} \sum_n \mathbb{E}[\mathbf{y}_n \mathbf{y}_n^\top] - \boldsymbol{\mu}^t (\boldsymbol{\mu}^t)^\top \quad (6.164)$$

Thus we see that EM is *not* equivalent to simply replacing variables by their expectations and applying the standard MLE formula; that would ignore the posterior variance and would result in an incorrect estimate. Instead we must compute the expectation of the sufficient statistics, and plug that into the usual equation for the MLE.

6.5.4.3 Initialization

To get the algorithm started, we can compute the MLE based on those rows of the data matrix that are fully observed. If there are no such rows, we can just estimate the diagonal terms of $\boldsymbol{\Sigma}$ using the observed marginal statistics. We are then ready to start EM.

6.5.4.4 Example

As an example of this procedure in action, let us consider an imputation problem, where we have $N = 100$ 10-dimensional data cases, which we assume to come from a Gaussian. We generate synthetic data where 50% of the observations are missing at random. First we fit the parameters using EM. Call the resulting parameters $\hat{\boldsymbol{\theta}}$. We can now use our model for predictions by computing $\mathbb{E}[\mathbf{z}_n | \mathbf{x}_n, \hat{\boldsymbol{\theta}}]$. Figure 6.8 indicates that the results obtained using the learned parameters are almost as good as with the true parameters. Not surprisingly, performance improves with more data, or as the fraction of missing data is reduced.

6.5.5 Example: robust linear regression using Student likelihood

In this section, we discuss how to use EM to fit a linear regression model that uses the Student distribution for its likelihood, instead of the more common Gaussian distribution, in order to achieve robustness, as first proposed in [Zel76]. More precisely, the likelihood is given by

$$p(y|\mathbf{x}, \mathbf{w}, \sigma^2, \nu) = \mathcal{T}(y|\mathbf{w}^\top \mathbf{x}, \sigma^2, \nu) \quad (6.165)$$

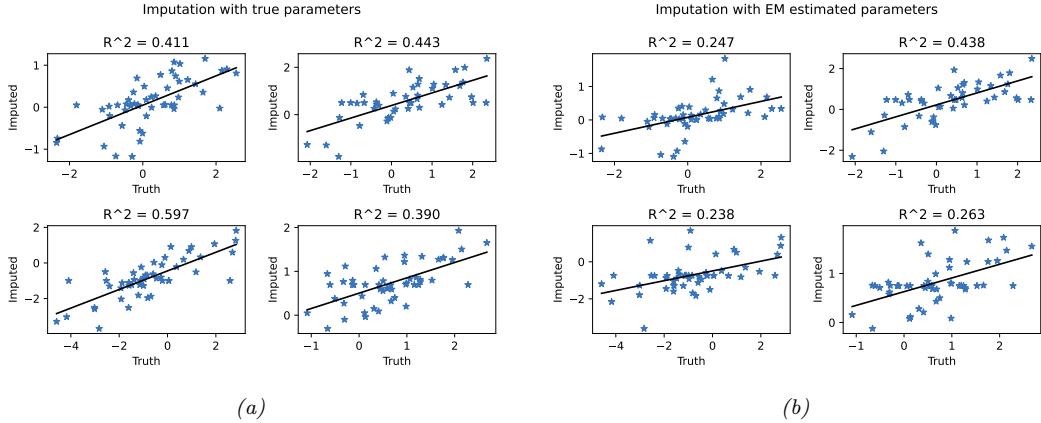


Figure 6.8: Illustration of data imputation using a multivariate Gaussian. (a) Scatter plot of true values vs imputed values using true parameters. (b) Same as (a), but using parameters estimated with EM. We just show the first four variables, for brevity. Generated by [gauss_imputation_em_demo.ipynb](#).

At first blush it may not be apparent how to do this, since there is no missing data, and there are no hidden variables. However, it turns out that we can introduce “artificial” hidden variables to make the problem easier to solve; this is a common trick. The key insight is that we can represent the Student distribution as a Gaussian scale mixture, as we discuss in Section 28.2.3.1.

We can apply the GSM version of the Student distribution to our problem by associating a latent scale $z_n \in \mathbb{R}_+$ with each example. The complete data log likelihood is therefore given by

$$\log p(\mathbf{y}, \mathbf{z} | \mathbf{X}, \mathbf{w}, \sigma^2, \nu) = \sum_n -\frac{1}{2} \log(2\pi z_n \sigma^2) - \frac{1}{2z_n \sigma^2} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \quad (6.166)$$

$$+ \left(\frac{\nu}{2} - 1 \right) \log(z_n) - z_n \frac{\nu}{2} + \text{const} \quad (6.167)$$

Ignoring terms not involving \mathbf{w} , and taking expectations, we have

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = - \sum_n \frac{\lambda_n^t}{2\sigma^2} (y_n - \mathbf{w}^T \mathbf{x}_n)^2 \quad (6.168)$$

where $\lambda_n^t \triangleq \mathbb{E}[1/z_n | y_n, \mathbf{x}_n, \mathbf{w}^t]$. We recognize this as a weighted least squares objective, with weight λ_n^t per datapoint.

We now discuss how to compute these weights. Using the results from Section 2.2.3.4, one can show that

$$p(z_n | y_n, \mathbf{x}_n, \boldsymbol{\theta}) = \text{IG}\left(\frac{\nu + 1}{2}, \frac{\nu + \delta_n}{2}\right) \quad (6.169)$$

where $\delta_n = \frac{(y_n - \mathbf{w}^T \mathbf{x}_n)^2}{\sigma^2}$ is the standardized residual. Hence

$$\lambda_n = \mathbb{E}[1/z_n] = \frac{\nu^t + 1}{\nu^t + \delta_n^t} \quad (6.170)$$

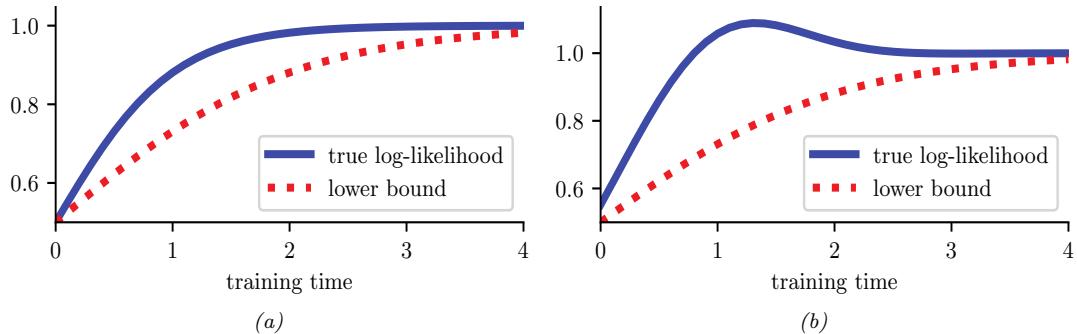


Figure 6.9: Illustration of possible behaviors of variational EM. (a) The lower bound increases at each iteration, and so does the likelihood. (b) The lower bound increases but the likelihood decreases. In this case, the algorithm is closing the gap between the approximate and true posterior. This can have a regularizing effect. Adapted from Figure 6 of [SJ96]. Generated by [var_em_bound.ipynb](#).

So if the residual δ_n^t is large, the point will be given low weight λ_n^t , which makes intuitive sense, since it is probably an outlier.

6.5.6 Extensions to EM

There are many variations and extensions of the EM algorithm, as discussed in [MK97]. We summarize a few of these below.

6.5.6.1 Variational EM

Suppose in the E step we pick $q_n^* = \operatorname{argmin}_{q_n \in \mathcal{Q}} D_{\text{KL}}(q_n \| p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta}))$. Because we are optimizing over the space of functions, this is called variational inference (see Section 10.1 for details). If the family of distributions \mathcal{Q} is rich enough to contain the true posterior, $q_n = p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta})$, then we can make the KL be zero. But in general, we might choose a more restrictive class for computational reasons. For example, we might use $q_n(\mathbf{z}_n) = \mathcal{N}(\mathbf{z}_n | \boldsymbol{\mu}_n, \operatorname{diag}(\boldsymbol{\sigma}_n))$ even if the true posterior is correlated.

The use of a restricted posterior family \mathcal{Q} inside the E step of EM is called **variational EM** [NH98a]. Unlike regular EM, variational EM is not guaranteed to increase the actual log likelihood itself (see Figure 6.9), but it does monotonically increase the variational lower bound. We can control the tightness of this lower bound by varying the variational family \mathcal{Q} ; in the limit in which $q_n = p_n$, corresponding to exact inference, we recover the same behavior as regular EM. See Section 10.1.3 for further discussion.

6.5.6.2 Hard EM

Suppose we use a degenerate posterior approximation in the context of variational EM, corresponding to a point estimate, $q(\mathbf{z} | \mathbf{x}_n) = \delta_{\hat{\mathbf{z}}_n}(\mathbf{z})$, where $\hat{\mathbf{z}}_n = \operatorname{argmax}_{\mathbf{z}} p(\mathbf{z} | \mathbf{x}_n)$. This is equivalent to **hard EM**, where we ignore uncertainty about \mathbf{z}_n in the E step.

The problem with this degenerate approach is that it is very prone to overfitting, since the number of latent variables is proportional to the number of datacases [WCS08].

6.5.6.3 Monte Carlo EM

Another approach to handling an intractable E step is to use a Monte Carlo approximation to the expected sufficient statistics. That is, we draw samples from the posterior, $\mathbf{z}_n^s \sim p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta}^t)$; then we compute the sufficient statistics for each completed vector, $(\mathbf{x}_n, \mathbf{z}_n^s)$; and finally we average the results. This is called **Monte Carlo EM** or **MCEM** [WT90; Nea12].

One way to draw samples is to use MCMC (see Chapter 12). However, if we have to wait for MCMC to converge inside each E step, the method becomes very slow. An alternative is to use stochastic approximation, and only perform “brief” sampling in the E step, followed by a partial parameter update. This is called **stochastic approximation EM** [DLM99] and tends to work better than MCEM.

6.5.6.4 Generalized EM

Sometimes we can perform the E step exactly, but we cannot perform the M step exactly. However, we can still monotonically increase the log likelihood by performing a “partial” M step, in which we merely increase the expected complete data log likelihood, rather than maximizing it. For example, we might follow a few gradient steps. This is called the **generalized EM** or **GEM** algorithm [MK07]. (This is an unfortunate term, since there are many ways to generalize EM, but it is the standard terminology.) For example, [Lan95a] proposes to perform one Newton-Raphson step:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{H}_t^{-1} \mathbf{g}_t \quad (6.171)$$

where $0 < \eta_t \leq 1$ is the step size, and

$$\mathbf{g}_t = \frac{\partial}{\partial \boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_t) |_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \quad (6.172)$$

$$\mathbf{H}_t = \frac{\partial^2}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_t) |_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \quad (6.173)$$

If $\eta_t = 1$, [Lan95a] calls this the **gradient EM algorithm**. However, it is possible to use a larger step size to speed up the algorithm, as in the **quasi-Newton EM algorithm** of [Lan95b]. This method also replaces the Hessian in Equation (6.173), which may not be negative definite (for non exponential family models), with a BFGS approximation. This ensures the overall algorithm is an ascent algorithm. Note, however, when the M step cannot be computed in closed form, EM loses some of its appeal over directly optimizing the marginal likelihood with a gradient based solver.

6.5.6.5 ECM algorithm

The **ECM** algorithm stands for “expectation conditional maximization”, and refers to optimizing the parameters in the M step sequentially, if they turn out to be dependent. The **ECME** algorithm, which stands for “ECM either” [LR95], is a variant of ECM in which we maximize the expected complete data log likelihood (the Q function) as usual, or the observed data log likelihood, during one or more of the conditional maximization steps. The latter can be much faster, since it ignores

the results of the E step, and directly optimizes the objective of interest. A standard example of this is when fitting the Student distribution. For fixed ν , we can update Σ as usual, but then to update ν , we replace the standard update of the form $\nu^{t+1} = \arg \max_{\nu} Q((\boldsymbol{\mu}^{t+1}, \Sigma^{t+1}, \nu), \boldsymbol{\theta}^t)$ with $\nu^{t+1} = \arg \max_{\nu} \log p(\mathcal{D} | \boldsymbol{\mu}^{t+1}, \Sigma^{t+1}, \nu)$. See [MK97] for more information.

6.5.6.6 Online EM

When dealing with large or streaming datasets, it is important to be able to learn online, as we discussed in Section 19.7.5. There are two main approaches to **online EM** in the literature. The first approach, known as **incremental EM** [NH98a], optimizes the lower bound $Q(\boldsymbol{\theta}, q_1, \dots, q_N)$ one q_n at a time; however, this requires storing the expected sufficient statistics for each data case.

The second approach, known as **stepwise EM** [SI00; LK09; CM09], is based on stochastic gradient descent. This optimizes a local upper bound on $\ell_n(\boldsymbol{\theta}) = \log p(\mathbf{x}_n | \boldsymbol{\theta})$ at each step. (See [Mai13; Mai15] for a more general discussion of stochastic and incremental bound optimization algorithms.)

6.6 Bayesian optimization

In this section, we discuss **Bayesian optimization** or **BayesOpt**, which is a model-based approach to black-box optimization, designed for the case where the objective function $f : \mathcal{X} \rightarrow \mathbb{R}$ is expensive to evaluate (e.g., if it requires running a simulation, or training and testing a particular neural net architecture).

Since the true function f is expensive to evaluate, we want to make as few function calls (i.e., make as few **queries** \mathbf{x} to the **oracle** f) as possible. This suggests that we should build a **surrogate function** (also called a **response surface model**) based on the data collected so far, $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : i = 1 : n\}$, which we can use to decide which point to query next. There is an inherent tradeoff between picking the point \mathbf{x} where we think $f(\mathbf{x})$ is large (we follow the convention in the literature and assume we are trying to maximize f), and picking points where we are uncertain about $f(\mathbf{x})$ but where observing the function value might help us improve the surrogate model. This is another instance of the exploration-exploitation dilemma.

In the special case where the domain we are optimizing over is finite, so $\mathcal{X} = \{1, \dots, A\}$, the BayesOpt problem becomes similar to the **best arm identification** problem in the bandit literature (Section 34.4). An important difference is that in bandits, we care about the cost of every action we take, whereas in optimization, we usually only care about the cost of the final solution we find. In other words, in bandits, we want to minimize cumulative regret, whereas in optimization we want to minimize simple or final regret.

Another related topic is **active learning**. Here the goal is to identify the whole function f with as few queries as possible, whereas in BayesOpt, the goal is just to identify the maximum of the function.

Bayesian optimization is a large topic, and we only give a brief overview below. For more details, see e.g., [Sha+16; Fra18; Gar23]. (See also <https://distill.pub/2020/bayesian-optimization/> for an interactive tutorial.)

6.6.1 Sequential model-based optimization

BayesOpt is an instance of a strategy known as sequential model-based optimization (**SMBO**) [HHLB11]. In this approach, we alternate between querying the function at a point, and updating our estimate of the surrogate based on the new data. More precisely, at each iteration n , we have a labeled dataset $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : i = 1 : n\}$, which records points \mathbf{x}_i that we have queried, and the corresponding function values, $y_i = f(\mathbf{x}_i) + \epsilon_i$, where ϵ_i is an optional noise term. We use this dataset to estimate a probability distribution over the true function f ; we will denote this by $p(f|\mathcal{D}_n)$. We then choose the next point to query \mathbf{x}_{n+1} using an **acquisition function** $\alpha(\mathbf{x}; \mathcal{D}_n)$, which computes the expected utility of querying \mathbf{x} . (We discuss acquisition functions in Section 6.6.3). After we observe $y_{n+1} = f(\mathbf{x}_{n+1}) + \epsilon_{n+1}$, we update our beliefs about the function, and repeat. See Algorithm 6.5 for some pseudocode.

Algorithm 6.5: Bayesian optimization

- 1 Collect initial dataset $\mathcal{D}_0 = \{(\mathbf{x}_i, y_i)\}$ from random queries \mathbf{x}_i or a space-filling design
 - 2 Initialize model by computing $p(f|\mathcal{D}_0)$
 - 3 **for** $n = 1, 2, \dots$ *until convergence do*
 - 4 Choose next query point $\mathbf{x}_{n+1} = \operatorname{argmax}_{\mathbf{x} \in \mathcal{X}} \alpha(\mathbf{x}; \mathcal{D}_n)$
 - 5 Measure function value, $y_{n+1} = f(\mathbf{x}_{n+1}) + \epsilon_n$
 - 6 Augment dataset, $\mathcal{D}_{n+1} = \{\mathcal{D}_n, (\mathbf{x}_{n+1}, y_{n+1})\}$
 - 7 Update model by computing $p(f|\mathcal{D}_{n+1})$
-

This method is illustrated in Figure 6.10. The goal is to find the global optimum of the solid black curve. In the first row, we show the 2 previously queried points, x_1 and x_2 , and their corresponding function values. $y_1 = f(x_1)$ and $y_2 = f(x_2)$. Our uncertainty about the value of f at those locations is 0 (if we assume no observation noise), as illustrated by the posterior credible interval (shaded blue arc) becoming “pinched”. Consequently the acquisition function (shown in green at the bottom) also has value 0 at those previously queried points. The red triangle represents the maximum of the acquisition function, which becomes our next query, x_3 . In the second row, we show the result of observing $y_3 = f(x_3)$; this further reduces our uncertainty about the shape of the function. In the third row, we show the result of observing $y_4 = f(x_4)$. This process repeats until we run out of time, or until we are confident there are no better unexplored points to query.

The two main “ingredients” that we need to provide to a BayesOpt algorithm are (1) a way to represent and update the posterior surrogate $p(f|\mathcal{D}_n)$, and (2) a way to define and optimize the acquisition function $\alpha(\mathbf{x}; \mathcal{D}_n)$. We discuss both of these topics below.

6.6.2 Surrogate functions

In this section, we discuss ways to represent and update the posterior over functions, $p(f|\mathcal{D}_n)$.

6.6.2.1 Gaussian processes

In BayesOpt, it is very common to use a Gaussian process or GP for our surrogate. GPs are explained in detail in Chapter 18, but the basic idea is that they represent $p(f(\mathbf{x})|\mathcal{D}_n)$ as a Gaussian,

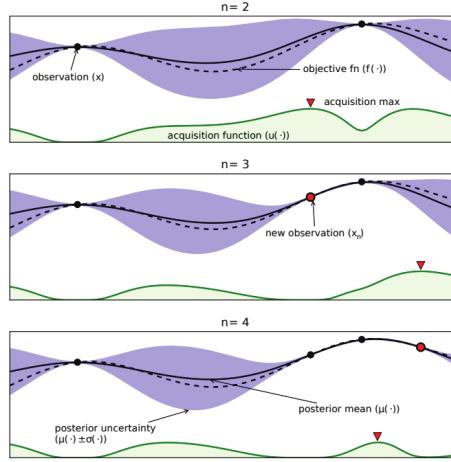


Figure 6.10: Illustration of sequential Bayesian optimization over three iterations. The rows correspond to a training set of size $t = 2, 3, 4$. The dotted black line is the true, but unknown, function $f(x)$. The solid black line is the posterior mean, $\mu(x)$. The shaded blue intervals are the 95% credible interval derived from $\mu(x)$ and $\sigma(x)$. The solid black dots correspond to points whose function value has already been computed, i.e., x_n for which $f(x_n)$ is known. The green curve at the bottom is the acquisition function. The red dot is the proposed next point to query, which is the maximum of the acquisition function. From Figure 1 of [Sha+16]. Used with kind permission of Nando de Freitas.

$p(f(\mathbf{x})|\mathcal{D}_n) = \mathcal{N}(f|\mu_n(\mathbf{x}), \sigma_n^2(\mathbf{x}))$, where $\mu_n(\mathbf{x})$ and $\sigma_n(\mathbf{x})$ are functions that can be derived from the training data $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : i = 1 : n\}$ using a simple closed-form equation. The GP requires specifying a kernel function $\mathcal{K}_{\theta}(\mathbf{x}, \mathbf{x}')$, which measures similarities between input points \mathbf{x}, \mathbf{x}' . The intuition is that if two inputs are similar, so $\mathcal{K}_{\theta}(\mathbf{x}, \mathbf{x}')$ is large, then the corresponding function values are also likely to be similar, so $f(\mathbf{x})$ and $f(\mathbf{x}')$ should be positively correlated. This allows us to interpolate the function between the labeled training points; in some cases, it also lets us extrapolate beyond them.

GPs work well when we have little training data, and they support closed form Bayesian updating. However, exact updating takes $O(N^3)$ for N samples, which becomes too slow if we perform many function evaluations. There are various methods (Section 18.5.3) for reducing this to $O(NM^2)$ time, where M is a parameter we choose, but this sacrifices some of the accuracy.

In addition, the performance of GPs depends heavily on having a good kernel. We can estimate the kernel parameters θ by maximizing the marginal likelihood, as discussed in Section 18.6.1. However, since the sample size is small (by assumption), we can often get better performance by marginalizing out θ using approximate Bayesian inference methods, as discussed in Section 18.6.2. See e.g., [WF16] for further details.

6.6.2.2 Bayesian neural networks

A natural alternative to GPs is to use a parametric model. If we use linear regression, we can efficiently perform exact Bayesian inference, as shown in Section 15.2. If we use a nonlinear model,

such as a DNN, we need to use approximate inference methods. We discuss Bayesian neural networks in detail in Chapter 17. For their application to BayesOpt, see e.g., [Spr+16; PPR22; Kim+22].

6.6.2.3 Other models

We are free to use other forms of regression model. [HHLB11] use an ensemble of random forests; such models can easily handle conditional parameter spaces, as we discuss in Section 6.6.4.2, although bootstrapping (which is needed to get uncertainty estimates) can be slow.

6.6.3 Acquisition functions

In BayesOpt, we use an **acquisition function** (also called a **merit function**) to evaluate the expected utility of each possible point we could query: $\alpha(\mathbf{x}|\mathcal{D}_n) = \mathbb{E}_{p(y|\mathbf{x}, \mathcal{D}_n)}[U(\mathbf{x}, y; \mathcal{D}_n)]$, where $y = f(\mathbf{x})$ is the unknown value of the function at point \mathbf{x} , and $U()$ is a utility function. Different utility functions give rise to different acquisition functions, as we discuss below. We usually choose functions so that the utility of picking a point that has already been queried is small (or 0, in the case of noise-free observations), in order to encourage exploration.

6.6.3.1 Probability of improvement

Let us define $M_n = \max_{i=1}^n y_i$ to be the best value observed so far (known as the **incumbent**). (If the observations are noisy, using the highest mean value $\max_i \mathbb{E}_{p(f|\mathcal{D}_n)}[f(\mathbf{x}_i)]$ is a reasonable alternative [WF16].) Then we define the utility of some new point \mathbf{x} using $U(\mathbf{x}, y; \mathcal{D}_n) = \mathbb{I}(y > M_n)$. This gives reward iff the new value is better than the incumbent. The corresponding acquisition function is then given by the expected utility, $\alpha_{PI}(\mathbf{x}; \mathcal{D}_n) = p(f(\mathbf{x}) > M_n | \mathcal{D}_n)$. This is known as the **probability of improvement** [Kus64]. If $p(f|\mathcal{D}_n)$ is a GP, then this quantity can be computed in closed form, as follows:

$$\alpha_{PI}(\mathbf{x}; \mathcal{D}_n) = p(f(\mathbf{x}) > M_n | \mathcal{D}_n) = \Phi(\gamma_n(\mathbf{x}, M_n)) \quad (6.174)$$

where Φ is the cdf of the $\mathcal{N}(0, 1)$ distribution and

$$\gamma_n(\mathbf{x}, \tau) = \frac{\mu_n(\mathbf{x}) - \tau}{\sigma_n(\mathbf{x})} \quad (6.175)$$

6.6.3.2 Expected improvement

The problem with PI is that all improvements are considered equally good, so the method tends to exploit quite aggressively [Jon01]. A common alternative takes into account the amount of improvement by defining $U(\mathbf{x}, y; \mathcal{D}_n) = (y - M_n)\mathbb{I}(y > M_n)$ and

$$\alpha_{EI}(\mathbf{x}; \mathcal{D}_n) = \mathbb{E}_{\mathcal{D}_n}[U(\mathbf{x}, y)] = \mathbb{E}_{\mathcal{D}_n}[(f(\mathbf{x}) - M_n)\mathbb{I}(f(\mathbf{x}) > M_n)] \quad (6.176)$$

This acquisition function is known as the **expected improvement (EI)** criterion [Moc+96]. In the case of a GP surrogate, this has the following closed form expression:

$$\alpha_{EI}(\mathbf{x}; \mathcal{D}_n) = (\mu_n(\mathbf{x}) - M_n)\Phi(\gamma) + \sigma_n(\mathbf{x})\phi(\gamma) = \sigma_n(\mathbf{x})[\gamma_n\Phi(\gamma) + \phi(\gamma)] \quad (6.177)$$

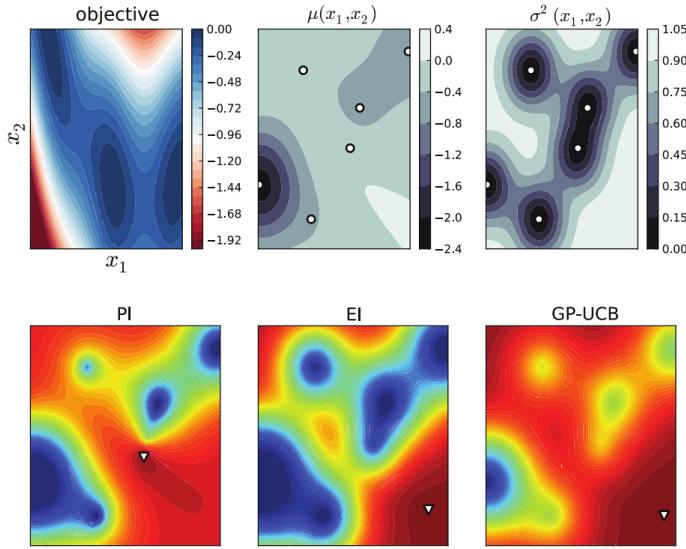


Figure 6.11: The first row shows the objective function, (the Branin function defined on \mathbb{R}^2), and its posterior mean and variance using a GP estimate. White dots are the observed data points. The second row shows 3 different acquisition functions (probability of improvement, expected improvement, and upper confidence bound); the white triangles are the maxima of the corresponding acquisition functions. From Figure 6 of [BCF10]. Used with kind permission of Nando de Freitas.

where $\phi()$ is the pdf of the $\mathcal{N}(0, 1)$ distribution, Φ is the cdf, and $\gamma = \gamma_n(\mathbf{x}, M_n)$. The first term encourages exploitation (evaluating points with high mean) and the second term encourages exploration (evaluating points with high variance). This is illustrated in Figure 6.10.

If we cannot compute the predictive variance analytically, but can draw posterior samples, then we can compute a Monte Carlo approximation to the EI, as proposed in [Kim+22]:

$$\alpha_{EI}(\mathbf{x}; \mathcal{D}_n) \approx \frac{1}{S} \sum_{s=1}^S \max(\mu_n^s(\mathbf{x}) - M_n, 0) \quad (6.178)$$

6.6.3.3 Upper confidence bound (UCB)

An alternative approach is to compute an **upper confidence bound** or **UCB** on the function, at some confidence level β_n , and then to define the acquisition function as follows: $\alpha_{UCB}(\mathbf{x}; \mathcal{D}_n) = \mu_n(\mathbf{x}) + \beta_n \sigma_n(\mathbf{x})$. This is the same as in the contextual bandit setting, discussed in Section 34.4.5, except we are optimizing over $\mathbf{x} \in \mathcal{X}$, rather than a finite set of arms $a \in \{1, \dots, A\}$. If we use a GP for our surrogate, the method is known as **GP-UCB** [Sri+10].

6.6.3.4 Thompson sampling

We discuss **Thompson sampling** in Section 34.4.6 in the context of multiarmed bandits, where the state space is finite, $\mathcal{X} = \{1, \dots, A\}$, and the acquisition function $\alpha(a; \mathcal{D}_n)$ corresponds to the probability that arm a is the best arm. We can generalize this to real-valued input spaces \mathcal{X} using

$$\alpha(\mathbf{x}; \mathcal{D}_n) = \mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D}_n)} \left[\mathbb{I}\left(\mathbf{x} = \operatorname{argmax}_{\mathbf{x}'} f_{\boldsymbol{\theta}}(\mathbf{x}')\right) \right] \quad (6.179)$$

We can compute a single sample approximation to this integral by sampling $\tilde{\boldsymbol{\theta}} \sim p(\boldsymbol{\theta}|\mathcal{D}_n)$. We can then pick the optimal action as follows:

$$\mathbf{x}_{n+1} = \operatorname{argmax}_{\mathbf{x}} \alpha(\mathbf{x}; \mathcal{D}_n) = \operatorname{argmax}_{\mathbf{x}} \mathbb{I}\left(\mathbf{x} = \operatorname{argmax}_{\mathbf{x}'} f_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}')\right) = \operatorname{argmax}_{\mathbf{x}} f_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}) \quad (6.180)$$

In other words, we greedily maximize the sampled surrogate.

For continuous spaces, Thompson sampling is harder to apply than in the bandit case, since we can't directly compute the best “arm” \mathbf{x}_{n+1} from the sampled function. Furthermore, when using GPs, there are some subtle technical difficulties with sampling a function, as opposed to sampling the parameters of a parametric surrogate model (see [HLHG14] for discussion).

6.6.3.5 Entropy search

Since our goal in BayesOpt is to find $\mathbf{x}^* = \operatorname{argmax}_{\mathbf{x}} f(\mathbf{x})$, it makes sense to try to directly minimize our uncertainty about the location of \mathbf{x}^* , which we denote by $p_*(\mathbf{x}|\mathcal{D}_n)$. We will therefore define the utility as follows:

$$U(\mathbf{x}, y; \mathcal{D}_n) = \mathbb{H}(\mathbf{x}^*|\mathcal{D}_n) - \mathbb{H}(\mathbf{x}^*|\mathcal{D}_n \cup \{(\mathbf{x}, y)\}) \quad (6.181)$$

where $\mathbb{H}(\mathbf{x}^*|\mathcal{D}_n) = \mathbb{H}(p_*(\mathbf{x}|\mathcal{D}_n))$ is the entropy of the posterior distribution over the location of the optimum. This is known as the information gain criterion; the difference from the objective used in active learning is that here we want to gain information about \mathbf{x}^* rather than about f for all \mathbf{x} . The corresponding acquisition function is given by

$$\alpha_{ES}(\mathbf{x}; \mathcal{D}_n) = \mathbb{E}_{p(y|\mathbf{x}, \mathcal{D}_n)} [U(\mathbf{x}, y; \mathcal{D}_n)] = \mathbb{H}(\mathbf{x}^*|\mathcal{D}_n) - \mathbb{E}_{p(y|\mathbf{x}, \mathcal{D}_n)} [\mathbb{H}(\mathbf{x}^*|\mathcal{D}_n \cup \{(\mathbf{x}, y)\})] \quad (6.182)$$

This is known as **entropy search** [HS12].

Unfortunately, computing $\mathbb{H}(\mathbf{x}^*|\mathcal{D}_n)$ is hard, since it requires a probability model over the input space. Fortunately, we can leverage the symmetry of mutual information to rewrite the acquisition function in Equation (6.182) as follows:

$$\alpha_{PES}(\mathbf{x}; \mathcal{D}_n) = \mathbb{H}(y|\mathcal{D}_n, \mathbf{x}) - \mathbb{E}_{\mathbf{x}^*|\mathcal{D}_n} [\mathbb{H}(y|\mathcal{D}_n, \mathbf{x}, \mathbf{x}^*)] \quad (6.183)$$

where we can approximate the expectation from $p(\mathbf{x}^*|\mathcal{D}_n)$ using Thompson sampling. Now we just have to model uncertainty about the output space y . This is known as **predictive entropy search** [HLHG14].

6.6.3.6 Knowledge gradient

So far the acquisition functions we have considered are all greedy, in that they only look one step ahead. The **knowledge gradient** acquisition function, proposed in [FPD09], looks two steps ahead by considering the improvement we might expect to get if we query \mathbf{x} , update our posterior, and then exploit our knowledge by maximizing wrt our new beliefs. More precisely, let us define the best value we can find if we query one more point:

$$V_{n+1}(\mathbf{x}, y) = \max_{\mathbf{x}'} \mathbb{E}_{p(f|\mathbf{x}, y, \mathcal{D}_n)} [f(\mathbf{x}')] \quad (6.184)$$

$$V_{n+1}(\mathbf{x}) = \mathbb{E}_{p(y|\mathbf{x}, \mathcal{D}_n)} [V_{n+1}(\mathbf{x}, y)] \quad (6.185)$$

We define the KG acquisition function as follows:

$$\alpha_{KG}(\mathbf{x}; \mathcal{D}_n) = \mathbb{E}_{\mathcal{D}_n} [(V_{n+1}(\mathbf{x}) - M_n) \mathbb{I}(V_{n+1}(\mathbf{x}) > M_n)] \quad (6.186)$$

Compare this to the EI function in Equation (6.176).) Thus we pick the point \mathbf{x}_{n+1} such that observing $f(\mathbf{x}_{n+1})$ will give us knowledge which we can then exploit, rather than directly trying to find a better point with better f value.

6.6.3.7 Optimizing the acquisition function

The acquisition function $\alpha(\mathbf{x})$ is often multimodal (see e.g., Figure 6.11), since it will be 0 at all the previously queried points (assuming noise-free observations). Consequently maximizing this function can be a hard subproblem in itself [WHD18; Rub+20].

In the continuous setting, it is common to use multirestart BFGS or grid search. We can also use the cross-entropy method (Section 6.7.5), using mixtures of Gaussians [BK10] or VAEs [Fau+18] as the generative model over \mathbf{x} . In the discrete, combinatorial setting (e.g., when optimizing biological sequences), [Bel+19] use regularized evolution, (Section 6.7.3), and [Ang+20] use proximal policy optimization (Section 35.3.4). Many other combinations are possible.

6.6.4 Other issues

There are many other issues that need to be tackled when using Bayesian optimization, a few of which we briefly mention below.

6.6.4.1 Parallel (batch) queries

In some cases, we want to query the objective function at multiple points in parallel; this is known as **batched Bayesian optimization**. Now we need to optimize over a set of possible queries, which is computationally even more difficult than the regular case. See [WHD18; DBB20] for some recent papers on this topic.

6.6.4.2 Conditional parameters

BayesOpt is often applied to hyper-parameter optimization. In many applications, some hyperparameters are only well-defined if other ones take on specific values. For example, suppose we are trying to automatically tune a classifier, as in the **Auto-Sklearn** system [Feu+15], or the **Auto-WEKA**

system [Kot+17]. If the method chooses to use a neural network, it also needs to specify the number of layers, and number of hidden units per layer; but if it chooses to use a decision tree, it instead should specify different hyperparameters, such as the maximum tree depth.

We can formalize such problems by defining the search space in terms of a tree or DAG (directed acyclic graph), where different subsets of the parameters are defined at each leaf. Applying GPs to this setting requires non-standard kernels, such as those discussed in [Swe+13; Jen+17]. Alternatively, we can use other forms of Bayesian regression, such as ensembles of random forests [HHLB11], which can easily handle conditional parameter spaces.

6.6.4.3 Multifidelity surrogates

In some cases, we can construct surrogate functions with different levels of accuracy, each of which may take variable amounts of time to compute. In particular, let $f(\mathbf{x}, s)$ be an approximation to the true function at \mathbf{x} with fidelity s . The goal is to solve $\max_{\mathbf{x}} f(\mathbf{x}, 0)$ by observing $f(\mathbf{x}, s)$ at a sequence of (\mathbf{x}_i, s_i) values, such that the total cost $\sum_{i=1}^n c(s_i)$ is below some budget. For example, in the context of hyperparameter selection, s may control how long we run the parameter optimizer for, or how large the validation set is.

In addition to choosing what fidelity to use for an experiment, we may choose to terminate expensive trials (queries) early, if the results of their cheaper proxies suggest they will not be worth running to completion (see e.g., [Str19; Li+17c; FKH17]). Alternatively, we may choose to resume an earlier aborted run, to collect more data on it, as in the **freeze-thaw algorithm** [SSA14].

6.6.4.4 Constraints

If we want to maximize a function subject to known constraints, we can simply build the constraints into the acquisition function. But if the constraints are unknown, we need to estimate the support of the feasible set in addition to estimating the function. In [GSA14], they propose the weighted EI criterion, given by $\alpha_{wEI}(\mathbf{x}; \mathcal{D}_n) = \alpha_{EI}(\mathbf{x}; \mathcal{D}_n)h(\mathbf{x}; \mathcal{D}_n)$, where $h(\mathbf{x}; \mathcal{D}_n)$ is a GP with a Bernoulli observation model that specifies if \mathbf{x} is feasible or not. Of course, other methods are possible. For example, [HL+16b] propose a method based on predictive entropy search.

6.7 Derivative-free optimization

Derivative-free optimization or **DFO** refers to a class of techniques for optimizing functions without using derivatives. This is useful for blackbox function optimization as well as discrete optimization. If the function is expensive to evaluate, we can use Bayesian optimization (Section 6.6). If the function is cheap to evaluate, we can use stochastic local search methods or evolutionary search methods, as we discuss below.

6.7.1 Local search

In this section, we discuss heuristic optimization algorithms that try to find the global maximum in a discrete, unstructured search space. These algorithms replace the local gradient based update, which has the form $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \mathbf{d}_t$, with the following discrete analog:

$$\mathbf{x}_{t+1} = \underset{\mathbf{x} \in \text{nbr}(\mathbf{x}_t)}{\operatorname{argmax}} \mathcal{L}(\mathbf{x}) \quad (6.187)$$

where $\text{nbr}(\mathbf{x}_t) \subseteq \mathcal{X}$ is the set of **neighbors** of \mathbf{x}_t . This is called **hill climbing**, **steepest ascent**, or **greedy search**.

If the “neighborhood” of a point contains the entire space, Equation (6.187) will return the global optimum in one step, but usually such a global neighborhood is too large to search exhaustively. Consequently we usually define local neighborhoods. For example, consider the **8-queens problem**. Here the goal is to place queens on an 8×8 chessboard so that they don’t attack each other (see Figure 6.14). The state space has the form $\mathcal{X} = 64^8$, since we have to specify the location of each queen on the grid. However, due to the constraints, there are only $8^8 \approx 17M$ feasible states. We define the neighbors of a state to be all possible states generated by moving a single queen to another square in the same column, so each node has $8 \times 7 = 56$ neighbors. According to [RN10, p.123], if we start at a randomly generated 8-queens state, steepest ascent gets stuck at a local maximum 86% of the time, so it only solves 14% of problem instances. However, it is fast, taking an average of 4 steps when it succeeds and 3 when it gets stuck.

In the sections below, we discuss slightly smarter algorithms that are less likely to get stuck in local maxima.

6.7.1.1 Stochastic local search

Hill climbing is greedy, since it picks the best point in its local neighborhood, by solving Equation (6.187) exactly. One way to reduce the chance of getting stuck in local maxima is to approximately maximize this objective at each step. For example, we can define a probability distribution over the uphill neighbors, proportional to how much they improve, and then sample one at random. This is called **stochastic hill climbing**. If we gradually decrease the entropy of this probability distribution (so we become greedier over time), we get a method called simulated annealing, which we discuss in Section 12.9.1.

Another simple technique is to use greedy hill climbing, but then whenever we reach a local maximum, we start again from a different random starting point. This is called **random restart hill climbing**. To see the benefit of this, consider again the 8-queens problem. If each hill-climbing search has a probability of $p \approx 0.14$ of success, then we expect to need $R = 1/p \approx 7$ restarts until we find a valid solution. The expected number of total steps can be computed as follows. Let $N_1 = 4$ be the average number of steps for successful trials, and $N_0 = 3$ be the average number of steps for failures. Then the total number of steps on average is $N_1 + (R - 1)N_0 = 4 + 6 \times 3 = 22$. Since each step is quick, the overall method is very fast. For example, it can solve an n -queens problem with $n=1M$ in under a minute.

Of course, solving the n -queens problem is not the most useful task in practice. However, it is typical of several real-world **boolean satisfiability problems**, which arise in problems ranging from AI planning to model checking (see e.g., [SLM92]). In such problems, simple **stochastic local search (SLS)** algorithms of the kind we have discussed work surprisingly well (see e.g., [HS05]).

6.7.1.2 Tabu search

Hill climbing will stop as soon as it reaches a local maximum or a plateau. Obviously one can perform a random restart, but this would ignore all the information that had been gained up to this point. A more intelligent alternative is called **tabu search** [GL97]. This is like hill climbing, except it allows moves that decrease (or at least do not increase) the scoring function, provided the move is to a new

Algorithm 6.6: Tabu search.

```

1  $t := 0$  // counts iterations
2  $c := 0$  // counts number of steps with no progress
3 Initialize  $\mathbf{x}_0$ 
4  $\mathbf{x}^* := \mathbf{x}_0$  // current best incumbent
5 while  $c < c_{\max}$  do
6    $\mathbf{x}_{t+1} = \operatorname{argmax}_{\mathbf{x} \in \text{nbr}(\mathbf{x}_t) \setminus \{\mathbf{x}_{t-\tau}, \dots, \mathbf{x}_{t-1}\}} f(\mathbf{x})$ 
7   if  $f(\mathbf{x}_{t+1}) > f(\mathbf{x}^*)$  then
8      $\mathbf{x}^* := \mathbf{x}_{t+1}$ 
9      $c := 0$ 
10   else
11      $c := c + 1$ 
12    $t := t + 1$ 
13 return  $\mathbf{x}^*$ 

```

state that has not been seen before. We can enforce this by keeping a tabu list which tracks the τ most recently visited states. This forces the algorithm to explore new states, and increases the chances of escaping from local maxima. We continue to do this for up to c_{\max} steps (known as the “tabu tenure”). The pseudocode can be found in Algorithm 6.6. (If we set $c_{\max} = 1$, we get greedy hill climbing.)

For example, consider what happens when tabu search reaches a hill top, \mathbf{x}_t . At the next step, it will move to one of the neighbors of the peak, $\mathbf{x}_{t+1} \in \text{nbr}(\mathbf{x}_t)$, which will have a lower score. At the next step, it will move to the neighbor of the previous step, $\mathbf{x}_{t+2} \in \text{nbr}(\mathbf{x}_{t+1})$; the tabu list prevents it cycling back to \mathbf{x}_t (the peak), so it will be forced to pick a neighboring point at the same height or lower. It continues in this way, “circling” the peak, possibly being forced downhill to a lower level-set (an inverse **basin flooding** operation), until it finds a ridge that leads to a new peak, or until it exceeds a maximum number of non-improving moves.

According to [RN10], p.123], tabu search increases the percentage of 8-queens problems that can be solved from 14% to 94%, although this variant takes an average of 21 steps for each successful instance and 64 steps for each failed instance.

6.7.1.3 Random search

A surprisingly effective strategy in problems where we know nothing about the objective is to use **random search**. In this approach, each iterate \mathbf{x}_{t+1} is chosen uniformly at random from \mathcal{X} . This should always be tried as a baseline.

In [BB12], they applied this technique to the problem of hyper-parameter optimization for some ML models, where the objective is performance on a validation set. In their examples, the search space is continuous, $\Theta = [0, 1]^D$. It is easy to sample from this at random. The standard alternative approach is to quantize the space into a fixed set of values, and then to evaluate them all; this is known as **grid search**. (Of course, this is only feasible if the number of dimensions D is small.) They found that random search outperformed grid search. The intuitive reason for this is that many

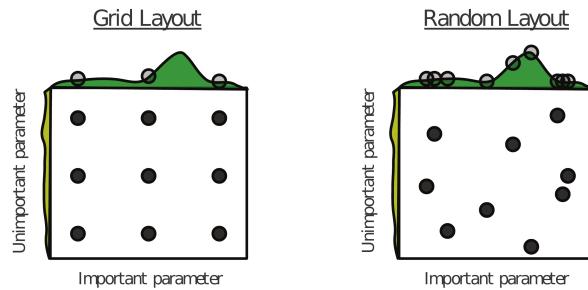


Figure 6.12: Illustration of grid search (left) vs random search (right). From Figure 1 of [BB12]. Used with kind permission of James Bergstra.

hyper-parameters do not make much difference to the objective function, as illustrated in Figure 6.12. Consequently it is a waste of time to place a fine grid along such unimportant dimensions.

RS has also been used to optimize the parameters of MDP policies, where the objective has the form $f(\boldsymbol{x}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{x}}} [R(\tau)]$ is the expected reward of trajectories generated by using a policy with parameters \boldsymbol{x} . For policies with few free parameters, RS can outperform more sophisticated reinforcement learning methods described in Chapter 35, as shown in [MGR18]. In cases where the policy has a large number of parameters, it is sometimes possible to project them to a lower dimensional random subspace, and perform optimization (either grid search or random search) in this subspace [Li+18a].

6.7.2 Simulated annealing

Simulated annealing [KJV83; LA87] is a **stochastic local search** algorithm (Section 6.7.1.1) that attempts to find the global minimum of a black-box function $\mathcal{E}(\boldsymbol{x})$, where $\mathcal{E}()$ is known as the **energy function**. The method works by converting the energy to an (unnormalized) probability distribution over states by defining $p(\boldsymbol{x}) = \exp(-\mathcal{E}(\boldsymbol{x}))$, and then using a variant of the **Metropolis-Hastings** algorithm to sample from a set of probability distributions, designed so that at the final step, the method samples from one of the modes of the distribution, i.e., it finds one of the most likely states, or lowest energy states. This approach can be used for both discrete and continuous optimization. See Section 12.9.1 for more details.

6.7.3 Evolutionary algorithms

Stochastic local search (SLS) maintains a single “best guess” at each step, \boldsymbol{x}_t . If we run this for T steps, and restart K times, the total cost is TK . A natural alternative is to maintain a set or **population** of K good candidates, \mathcal{S}_t , which we try to improve at each step. This is called an **evolutionary algorithm (EA)**. If we run this for T steps, it also takes TK time; however, it can often get better results than multi-restart SLS, since the search procedure explores more of the space in parallel, and information from different members of the population can be shared. Many versions of EA are possible, as we discuss below.

Since EA algorithms draw inspiration from the biological process of evolution, they also borrow

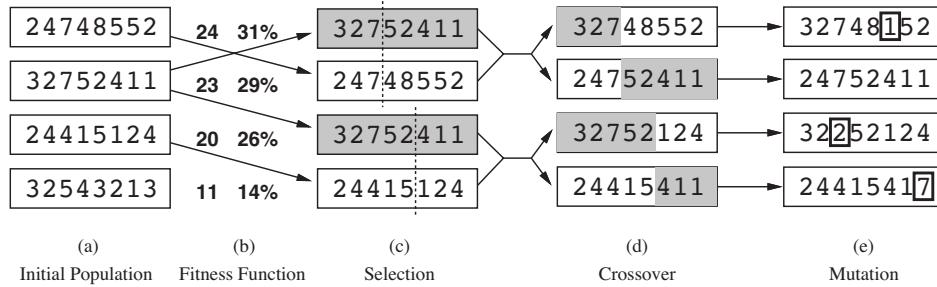


Figure 6.13: Illustration of a genetic algorithm applied to the 8-queens problem. (a) Initial population of 4 strings. (b) We rank the members of the population by fitness, and then compute their probability of mating. Here the integer numbers represent the number of nonattacking pairs of queens, so the global maximum has a value of 28. We pick an individual θ with probability $p(\theta) = \mathcal{L}(\theta)/Z$, where $Z = \sum_{\theta \in P} \mathcal{L}(\theta)$ sums the total fitness of the population. For example, we pick the first individual with probability $24/78 = 0.31$, the second with probability $23/78 = 0.29$, etc. In this example, we pick the first individual once, the second twice, the third one once, and the last one does not get to breed. (c) A split point on the “chromosome” of each parent is chosen at random. (d) The two parents swap their chromosome halves. (e) We can optionally apply pointwise mutation. From Figure 4.6 of [RN10]. Used with kind permission of Peter Norvig.

a lot of its terminology. The **fitness** of a member of the population is the value of the objective function (possibly normalized across population members). The members of the population at step $t + 1$ are called the **offspring**. These can be created by randomly choosing a **parent** from S_t and applying a random **mutation** to it. This is like asexual reproduction. Alternatively we can create an offspring by choosing two parents from S_t , and then combining them in some way to make a child, as in sexual reproduction; combining the parents is called **recombination**. (It is often followed by mutation.)

The procedure by which parents are chosen is called the **selection function**. In **truncation selection**, each parent is chosen from the fittest K members of the population (known as the **elite set**). In **tournament selection**, each parent is the fittest out of K randomly chosen members. In **fitness proportionate selection**, also called **roulette wheel selection**, each parent is chosen with probability proportional to its fitness relative to the others. We can also “kill off” the oldest members of the population, and then select parents based on their fitness; this is called **regularized evolution** [Rea+19]).

In addition to the selection rule for parents, we need to specify the recombination and mutation rules. There are many possible choices for these heuristics. We briefly mention a few of them below.

- In a **genetic algorithm (GA)** [Gol89; Hol92], we use mutation and a particular recombination method based on **crossover**. To implement crossover, we assume each individual is represented as a vector of integers or binary numbers, by analogy to **chromosomes**. We pick a split point along the chromosome for each of the two chosen parents, and then swap the strings, as illustrated in Figure 6.13.
- In **genetic programming** [Koz92], we use a tree-structured representation of individuals, instead of a bit string. This representation ensures that all crossovers result in valid children, as illustrated

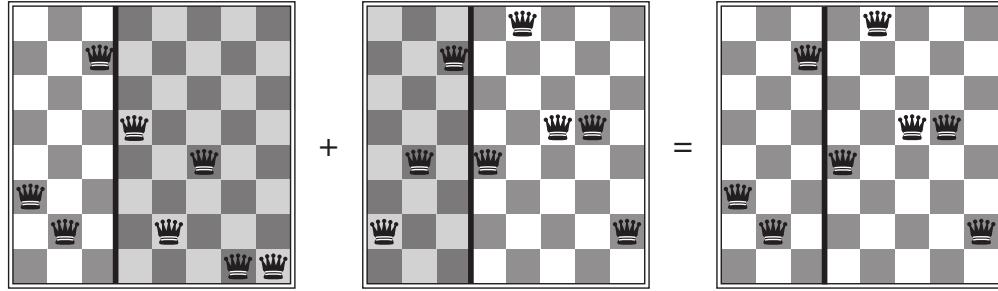


Figure 6.14: The 8-queens states corresponding to the first two parents in Figure 6.13(c) and their first child in Figure 6.13(d). We see that the encoding 32752411 means that the first queen is in row 3 (counting from the bottom left), the second queen is in row 2, etc. The shaded columns are lost in the crossover, but the unshaded columns are kept. From Figure 4.7 of [RN10]. Used with kind permission of Peter Norvig.

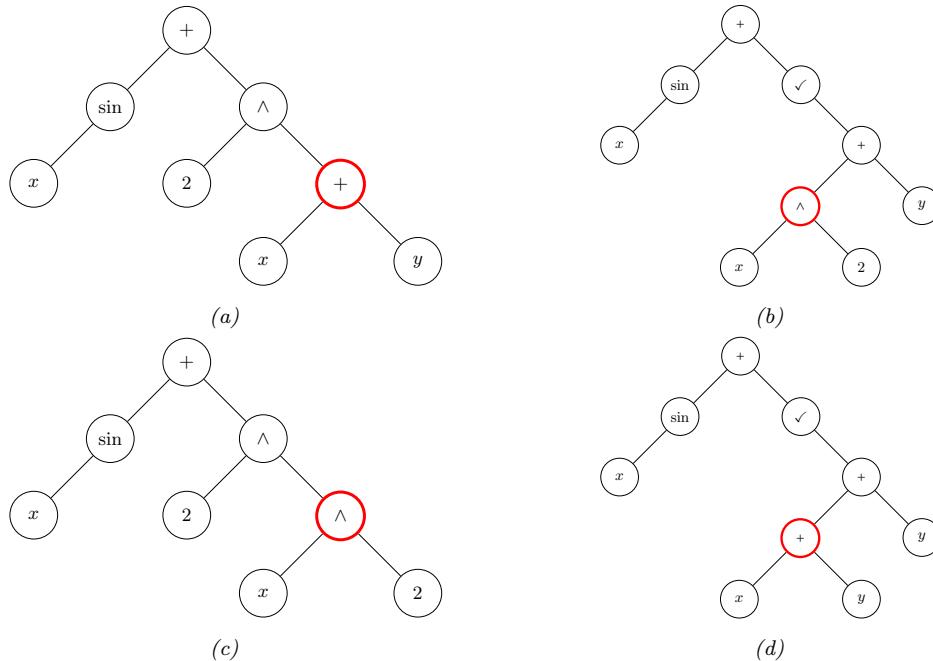


Figure 6.15: Illustration of crossover operator in a genetic program. (a-b) the two parents, representing $\sin(x) + (x+y)^2$ and $\sin(x) + \sqrt{x^2+y}$. The red circles denote the two crossover points. (c-d) the two children, representing $\sin(x) + (x^2)^2$ and $\sin(x) + \sqrt{x+y+y}$. Adapted from Figure 9.2 of [Mit97]

in Figure 6.15. Genetic programming can be useful for finding good programs as well as other structured objects, such as neural networks. In **evolutionary programming**, the structure of the tree is fixed and only the numerical parameters are evolved.

- In **surrogate assisted EA**, a surrogate function $\hat{f}(s)$ is used instead of the true objective function $f(s)$ in order to speed up the evaluation of members of the population (see [Jin11] for a survey). This is similar to the use of response surface models in Bayesian optimization (Section 6.6), except it does not deal with the explore-exploit tradeoff.
- In a **memetic algorithm** [MC03], we combine mutation and recombination with standard local search.

Evolutionary algorithms have been applied to a large number of applications, including training neural networks (this combination is known as **neuroevolution** [Sta+19]). An efficient JAX-based library for (neuro)-evolution can be found at <https://github.com/google/evojax>.

6.7.4 Estimation of distribution (EDA) algorithms

EA methods maintain a population of good candidate solutions, which can be thought of as an implicit (nonparametric) density model over states with high fitness. [BC95] proposed to “remove the genetics from GAs”, by explicitly learning a probabilistic model over the configuration space that puts its mass on high scoring solutions. That is, the population becomes the set of parameters of a generative model, θ .

One way to learn such a model is as follows. We start by creating a sample of $K' > K$ candidate solutions from the current model, $\mathcal{S}_t = \{\mathbf{x}_k \sim p(\mathbf{x}|\theta_t)\}$. We then rank the samples using the fitness function, and then pick the most promising subset \mathcal{S}_t^* of size K using a selection operator (this is known as **truncation selection**). Finally, we fit a new probabilistic model $p(\mathbf{x}|\theta_{t+1})$ to \mathcal{S}_t^* using maximum likelihood estimation. This is called the **estimation of distribution** or **EDA** algorithm (see e.g., [LL02; PSCP06; Hau+11; PHL12; Hu+12; San17; Bal17]).

Note that EDA is equivalent to minimizing the cross entropy between the empirical distribution defined by \mathcal{S}_t^* and the model distribution $p(\mathbf{x}|\theta_{t+1})$. Thus EDA is related to the **cross entropy method**, as described in Section 6.7.5, although CEM usually assumes the special case where $p(\mathbf{x}|\theta) = \mathcal{N}(\mathbf{x}|\mu, \Sigma)$. EDA is also closely related to the EM algorithm, as discussed in [Bro+20a].

As a simple example, suppose the configuration space is bit strings of length D , and the fitness function is $f(\mathbf{x}) = \sum_{d=1}^D x_d$, where $x_d \in \{0, 1\}$ (this is called the **one-max** function in the EA literature). A simple probabilistic model for this is a fully factored model of the form $p(\mathbf{x}|\theta) = \prod_{d=1}^D \text{Ber}(x_d|\theta_d)$. Using this model inside of DBO results in a method called univariate marginal distribution algorithm or **UMDA**.

We can estimate the parameters of the Bernoulli model by setting θ_d to the fraction of samples in \mathcal{S}_t^* that have bit d turned on. Alternatively, we can incrementally adjust the parameters. The population-based incremental learning (**PBIL**) algorithm [BC95] applies this idea to the factored Bernoulli model, resulting in the following update:

$$\hat{\theta}_{d,t+1} = (1 - \eta_t)\hat{\theta}_{d,t} + \eta_t \bar{\theta}_{d,t} \quad (6.188)$$

where $\bar{\theta}_{d,t} = \frac{1}{N_t} \sum_{k=1}^K \mathbb{I}(x_{k,d} = 1)$ is the MLE estimated from the $K = |\mathcal{S}_t^*|$ samples generated in the current iteration, and η_t is a learning rate.

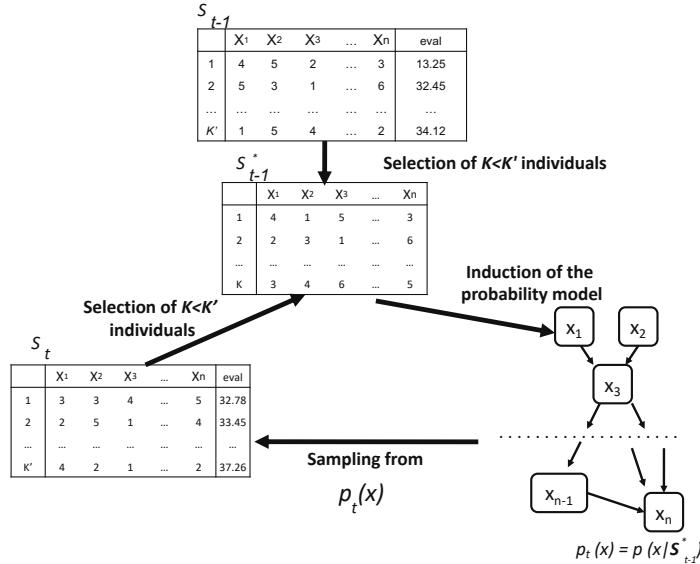


Figure 6.16: Illustration of the BOA algorithm (EDA applied to a generative model structured as a Bayes net). Adapted from Figure 3 of [PHL12].

It is straightforward to use more expressive probability models that capture dependencies between the parameters (these are known as **building blocks** in the EA literature). For example, in the case of real-valued parameters, we can use a multivariate Gaussian, $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The resulting method is called the **estimation of multivariate normal algorithm** or **EMNA**, [LL02]. (See also Section 6.7.5.)

For discrete random variables, it is natural to use probabilistic graphical models (Chapter 4) to capture dependencies between the variables. [BD97] learn a tree-structured graphical model using the Chow-Liu algorithm (Supplementary Section 30.2.1); [BJV97] is a special case of this where the graph is a tree. We can also learn more general graphical model structures (see e.g., [LL02]). We typically use a Bayes net (Section 4.2), since we can use ancestral sampling (Section 4.2.5) to easily generate samples; the resulting method is therefore called the **Bayesian optimization algorithm (BOA)** [PGCP00].⁵ The hierarchical BOA (**hBOA**) algorithm [PeI05] extends this by using decision trees and decision graphs to represent the local CPTs in the Bayes net (as in [CHM97]), rather than using tables. In general, learning the structure of the probability model for use in EDA is called **linkage learning**, by analogy to how genes can be linked together if they can be co-inherited as a building block.

We can also use deep generative models to represent the distribution over good candidates. For example, [CSF16] use denoising autoencoders and NADE models (Section 22.2), [Bal17] uses a DNN regressor which is then inverted using gradient descent on the inputs, [PRG17] uses RBMs

5. This should not be confused with the Bayesian optimization methods we discuss in Section 6.6, that use response surface modeling to model $p(f(\mathbf{x}))$ rather than $p(\mathbf{x}^*)$.

(Section 4.3.3.2), [GSM18] uses VAEs (Section 21.2), etc. Such models might take more data to fit (and therefore more function calls), but can potentially model the probability landscape more faithfully. (Whether that translates to better optimization performance is not clear, however.)

6.7.5 Cross-entropy method

The **cross-entropy method** [Rub97; RK04; Boe+05] is a special case of EDA (Section 6.7.4) in which the population is represented by a multivariate Gaussian. In particular, we set $\boldsymbol{\mu}_{t+1}$ and $\boldsymbol{\Sigma}_{t+1}$ to the empirical mean and covariance of \mathcal{S}_{t+1}^* , which are the top K samples. This is closely related to the SMC algorithm for sampling rare events discussed in Section 13.6.4.

The CEM is sometimes used for model-based RL (Section 35.4), since it is simple and can find reasonably good optima of multimodal objectives. It is also sometimes used inside of Bayesian optimization (Section 6.6), to optimize the multi-modal acquisition function (see [BK10]).

6.7.5.1 Differentiable CEM

The **differentiable CEM** method of [AY19] replaces the top K operator with a soft, differentiable approximation, which allows the optimizer to be used as part of an end-to-end differentiable pipeline. For example, we can use this to create a differentiable model predictive control (MPC) algorithm (Section 35.4.1), as described in Section 35.4.5.2.

The basic idea is as follows. Let $\mathcal{S}_t = \{\mathbf{x}_{t,i} \sim p(\mathbf{x}|\boldsymbol{\theta}_t) : i = 1 : K'\}$ represent the current population, with fitness values $v_{t,i} = f(\mathbf{x}_{t,i})$. Let $v_{t,K}^*$ be the K 'th smallest value. In CEM, we compute the set of top K samples, $\mathcal{S}_t^* = \{i : v_{t,i} \geq v_{t,K}^*\}$, and then update the model based on these: $\boldsymbol{\theta}_{t+1} = \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{i \in \mathcal{S}_t} p_t(i) \log p(\mathbf{x}_{t,i}|\boldsymbol{\theta})$, where $p_t(i) = \mathbb{I}(i \in \mathcal{S}_t^*) / |\mathcal{S}_t^*|$. In the differentiable version, we replace the sparse distribution \mathbf{p}_t with the “soft” dense distribution $\mathbf{q}_t = \Pi(\mathbf{p}_t; \tau, K)$, where

$$\Pi(\mathbf{p}; \tau, K) = \underset{\mathbf{0} \leq \mathbf{q} \leq \mathbf{1}}{\operatorname{argmin}} -\mathbf{p}^\top \mathbf{q} - \tau \mathbb{H}(\mathbf{q}) \quad \text{s.t. } \mathbf{1}^\top \mathbf{q} = K \tag{6.189}$$

projects the distribution \mathbf{p} onto the polytope of distributions which sum to K . (Here $\mathbb{H}(\mathbf{q}) = -\sum_i q_i \log(q_i) + (1 - q_i) \log(1 - q_i)$ is the entropy, and $\tau > 0$ is a temperature parameter.) This projection operator (and hence the whole DCEM algorithm) can be backpropagated through using implicit differentiation [AKZK19].

6.7.6 Evolutionary strategies

Evolution strategies [Wie+14] are a form of distribution-based optimization in which the distribution over the population is represented by a Gaussian, $p(\mathbf{x}|\boldsymbol{\theta}_t)$ (see e.g., [Sal+17b]). Unlike CEM, the parameters are updated using gradient ascent applied to the expected value of the objective, rather than using MLE on a set of elite samples. More precisely, consider the smoothed objective $\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p(\mathbf{x}|\boldsymbol{\theta})} [f(\mathbf{x})]$. We can use the REINFORCE estimator (Section 6.3.4) to compute the gradient of this objective as follows:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p(\mathbf{x}|\boldsymbol{\theta})} [f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})] \tag{6.190}$$

This can be approximated by drawing Monte Carlo samples. We discuss how to compute this gradient below.

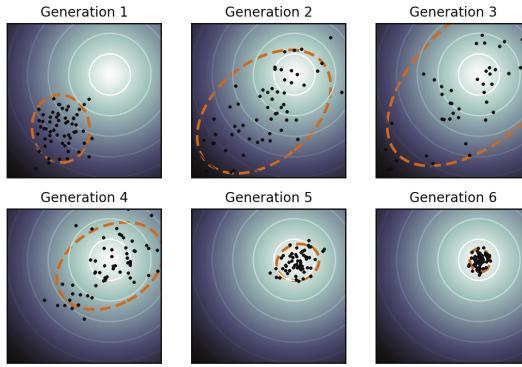


Figure 6.17: Illustration of the CMA-ES method applied to a simple 2d function. The dots represent members of the population, and the dashed orange ellipse represents the multivariate Gaussian. From <https://en.wikipedia.org/wiki/CMA-ES>. Used with kind permission of Wikipedia author Sentewolf.

6.7.6.1 Natural evolutionary strategies

If the probability model is in the exponential family, we can compute the natural gradient (Section 6.4), rather than the “vanilla” gradient, which can result in faster convergence. Such methods are called **natural evolution strategies** [Wie+14].

6.7.6.2 CMA-ES

The **CMA-ES** method of [Han16], which stands for “covariance matrix adaptation evolution strategy” is a kind of NES. It is very similar to CEM except it updates the parameters in a special way. In particular, instead of computing the new mean and covariance using unweighted MLE on the elite set, we attach weights to the elite samples based on their rank. We then set the new mean to the weighted MLE of the elite set.

The update equations for the covariance are more complex. In particular, “evolutionary paths” are also used to accumulate the search directions across successive generations, and these are used to update the covariance. It can be shown that the resulting updates approximate the natural gradient of $\mathcal{L}(\theta)$ without explicitly modeling the Fisher information matrix [Oll+17].

Figure 6.17 illustrates the method in action.

6.8 Optimal transport

This section is written by Marco Cuturi.

In this section, we focus on **optimal transport** theory, a set of tools that have been proposed, starting with work by [Mon81], to compare two probability distributions. We start from a simple example involving only matchings, and work from there towards various extensions.

6.8.1 Warm-up: matching optimally two families of points

Consider two families $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ and $(\mathbf{y}_1, \dots, \mathbf{y}_n)$, each consisting in $n > 1$ distinct points taken from a set \mathcal{X} . A *matching* between these two families is a bijective mapping that assigns to each point \mathbf{x}_i another point \mathbf{y}_j . Such an assignment can be encoded by pairing indices $(i, j) \in \{1, \dots, n\}^2$ such that they define a *permutation* σ in the symmetric group \mathcal{S}_n . With that convention and given a permutation σ , \mathbf{x}_i would be assigned to \mathbf{y}_{σ_i} , the σ_i 'th element in the second family.

Matchings costs. When matching a family with another, it is natural to consider the cost incurred when pairing any point \mathbf{x}_i with another point \mathbf{y}_j , for all possible pairs $(i, j) \in \{1, \dots, n\}^2$. For instance, \mathbf{x}_i might contain information on the current location of a taxi driver i , and \mathbf{y}_j that of a user j who has just requested a taxi; in that case, $C_{ij} \in \mathbb{R}$ may quantify the cost (in terms of time, fuel or distance) required for taxi driver i to reach user j . Alternatively, \mathbf{x}_i could represent a vector of skills held by a job seeker i and \mathbf{y}_j a vector quantifying desirable skills associated with a job posting j ; in that case C_{ij} could quantify the number of hours required for worker i to carry out job j . We will assume without loss of generality that the values C_{ij} are obtained by evaluating a cost function $c : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ on the pair $(\mathbf{x}_i, \mathbf{y}_j)$, namely $C_{ij} = c(\mathbf{x}_i, \mathbf{y}_j)$. In many applications of optimal transport, such cost functions have a geometric interpretation and are typically distance functions on \mathcal{X} as in Fig. 6.18, in which $\mathcal{X} = \mathbb{R}^2$, or as will be later discussed in Section 6.8.2.4.

Least-cost matchings. Equipped with a cost function c , the *optimal* matching (or assignment) problem is that of finding a permutation that reaches the smallest total cost, as defined by the function

$$\min_{\sigma} E(\sigma) = \sum_{i=1}^n c(\mathbf{x}_i, \mathbf{y}_{\sigma_i}). \quad (6.191)$$

The optimal matching problem is arguably one of the simplest combinatorial optimization problems, tackled as early as the 19th century [JB65]. Although a naive enumeration of all permutations would require evaluating objective E a total of $n!$ times, the Hungarian algorithm [Kuh55] was shown to provide the optimal solution in polynomial time [Mun57], and later refined to require in the worst case $O(n^3)$ operations.

6.8.2 From optimal matchings to Kantorovich and Monge formulations

The optimal matching problem is relevant to many applications, but it suffers from a few limitations. One could argue that most of the optimal transport literature arises from the necessity to overcome these limitations and extend (6.191) to more general settings. An obvious issue arises when the number of points available in both families is not the same. The second limitation arises when considering a continuous setting, namely when trying to match (or morph) two probability densities, rather than families of atoms (discrete measures).

6.8.2.1 Mass splitting

Suppose again that all points \mathbf{x}_i and \mathbf{y}_j describe skills, respectively held by a worker i and needed for a task j to be fulfilled in a factory. Since finding a matching is equivalent to finding a permutation in $\{1, \dots, n\}$, problem (6.191) cannot handle cases in which the number of workers is larger (or smaller) than the number of tasks. More problematically, the assumption that every single task is indivisible,

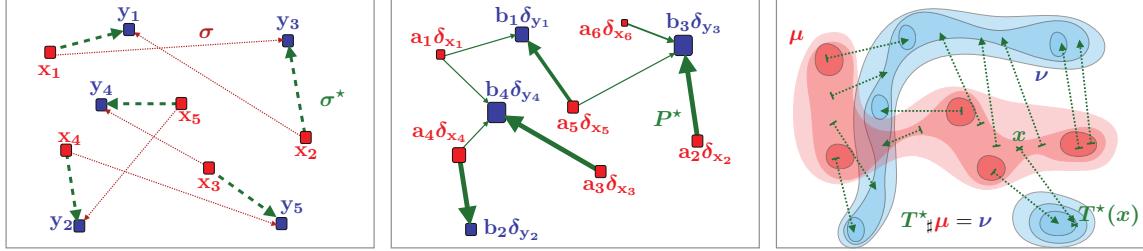


Figure 6.18: Left: Matching a family of 5 points to another is equivalent to considering a permutation in $\{1, \dots, n\}$. When to each pair $(\mathbf{x}_i, \mathbf{y}_j) \in \mathbb{R}^2$ is associated a cost equal to the distance $\|\mathbf{x}_i - \mathbf{y}_j\|$, the optimal matching problem involves finding a permutation σ that minimizes $\|\mathbf{x}_i - \mathbf{y}_{\sigma(i)}\|$ for i in $\{1, 2, 3, 4, 5\}$. Middle: The Kantorovich formulation of optimal transport generalizes optimal matchings, and arises when comparing discrete measures, that is, families of weighted points that do not necessarily share the same size but do share the same total mass. The relevant variable is a matrix P of size $n \times m$, which must satisfy row-sum and column-sum constraints, and which minimizes its dot product with matrix C_{ij} . Right: another direct extension of the matching problem lies when, intuitively, the number n of points that is described is such that the considered measures become continuous densities. In that setting, and unlike the Kantorovich setting, the goal is to seek a map $T : \mathcal{X} \rightarrow \mathcal{X}$ which, to any point x in the support of the input measure μ is associated a point $y = T(x)$ in the support of ν . The push-forward constraint $T_*\mu = \nu$ ensures that ν is recovered by applying map T to all points in the support of μ ; the optimal map T^* is that which minimizes the distance between x and $T(x)$, averaged over μ .

or that workers are only able to dedicate themselves to a single task, is hardly realistic. Indeed, certain tasks may require more (or less) dedication than that provided by a single worker, whereas some workers may only be able to work part-time, or, on the contrary, be willing to put in extra hours. The rigid machinery of permutations falls short of handling such cases, since permutations are by definition one-to-one associations. The Kantorovich formulation allows for *mass-splitting*, the idea that the effort provided by a worker or needed to complete a given task can be split. In practice, to each of the n workers is associated, in addition to \mathbf{x}_i , a positive number $\mathbf{a}_i > 0$. That number represents the amount of time worker i is able to provide. Similarly, we introduce numbers $\mathbf{b}_j > 0$ describing the amount of time needed to carry out each of the m tasks (n and m do not necessarily coincide). Worker i is therefore described as a pair $(\mathbf{a}_i, \mathbf{x}_i)$, mathematically equivalent to a *weighted Dirac measure* $\mathbf{a}_i \delta_{\mathbf{x}_i}$. The overall workforce available to the factory is described as a discrete measure $\sum_i \mathbf{a}_i \delta_{\mathbf{x}_i}$, whereas its tasks are described in $\sum_j \mathbf{b}_j \delta_{\mathbf{y}_j}$. If one assumes further that the factory has a balanced workload, namely that $\sum_i \mathbf{a}_i = \sum_j \mathbf{b}_j$, then the Kantorovich [Kan42] formulation of optimal transport is:

$$\text{OT}_C(\mathbf{a}, \mathbf{b}) \triangleq \min_{P \in \mathbf{R}_+^{n \times m}, P\mathbf{1}_n = \mathbf{a}, P^T\mathbf{1}_m = \mathbf{b}} \langle P, C \rangle \triangleq \sum_{i,j} P_{ij} C_{ij}. \quad (6.192)$$

The interpretation behind such matrices is simple: each coefficient P_{ij} describes an allocation of time for worker i to spend on task j . The i 'th row-sum must be equal to the total \mathbf{a}_i for the time constraint of worker i to be satisfied, whereas the j 'th column-sum must be equal to \mathbf{b}_j , reflecting that the time needed to complete task j has been budgeted.

6.8.2.2 Monge formulation and optimal push forward maps

By introducing mass-splitting, the Kantorovich formulation of optimal transport allows for a far more general comparison between discrete measures of different sizes and weights (middle plot of Fig. 6.18). Naturally, this flexibility comes with a downside: one can no longer associate to each point \mathbf{x}_i another point \mathbf{y}_j to which it is uniquely associated, as was the case with the classical matching problem. Interestingly, this property can be recovered in the limit where the measures become densities. Indeed, the Monge [Mon81] formulation of optimal transport allows us to recover precisely that property, on the condition (loosely speaking) that measure μ admits a density. In that setting, the analogous mathematical object guaranteeing that μ is mapped onto ν is that of **push forward** maps morphing μ to ν , namely maps T such that for any measurable set $A \subset \mathcal{X}$, $\mu(T^{-1}(A)) = \nu(A)$. When T is differentiable, and μ, ν have densities p and q wrt the Lebesgue measure in \mathbb{R}^d , this statement is equivalent, thanks to the change of variables formula, to ensuring almost everywhere that:

$$q(T(x)) = p(x)|J_T(x)|, \quad (6.193)$$

where $|J_T(x)|$ stands for the determinant of the Jacobian matrix of T evaluated at x .

Writing $T_\sharp\mu = \nu$ when T does satisfy these conditions, the Monge [Mon81] problem consists in finding the best map T that minimizes the average cost between \mathbf{x} and its displacement $T(\mathbf{x})$,

$$\inf_{T:T_\sharp\mu=\nu} \int_{\mathcal{X}} c(\mathbf{x}, T(\mathbf{x})) \mu(d\mathbf{x}). \quad (6.194)$$

T is therefore a map that pushes μ forwards to ν globally, but which results, on average, in the smallest average cost. While very intuitive, the Monge problem turns out to be extremely difficult to solve in practice, since it is non-convex. Indeed, one can easily check that the constraint $\{T_\sharp\mu = \nu\}$ is not convex, since one can easily find counter-examples for which $T_\sharp\mu = \nu$ and $T'_\sharp\nu$ yet $(\frac{1}{2}T + \frac{1}{2}T')_\sharp\mu \neq \nu$. Luckily, Kantorovich's approach also works for continuous measures, and yields a comparatively much simpler linear program.

6.8.2.3 Kantorovich formulation

The Kantorovich problem (6.192) can also be extended to a continuous setting: Instead of optimizing over a subset of matrices in $\mathbb{R}^{n \times m}$, consider $\Pi(\mu, \nu)$, the subset of joint probability distributions $\mathcal{P}(\mathcal{X} \times \mathcal{X})$ with marginals μ and ν , namely

$$\Pi(\mu, \nu) \triangleq \{\pi \in \mathcal{P}(\mathcal{X}^2) : \forall A \subset \mathcal{X}, \pi(A \times \mathcal{X}) = \mu(A) \text{ and } \pi(\mathcal{X} \times A) = \nu(A)\}. \quad (6.195)$$

Note that $\Pi(\mu, \nu)$ is not empty since it always contains the product measure $\mu \otimes \nu$. With this definition, the continuous formulation of (6.192) can be obtained as

$$\text{OT}_c(\mu, \nu) \triangleq \inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X}^2} c \, d\pi. \quad (6.196)$$

Notice that (6.196) subsumes directly (6.192), since one can check that they coincide when μ, ν are discrete measures, with respective probability weights \mathbf{a}, \mathbf{b} and locations $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ and $(\mathbf{y}_1, \dots, \mathbf{y}_m)$.

6.8.2.4 Wasserstein distances

When c is equal to a metric d exponentiated by an integer, the optimal value of the Kantorovich problem is called the Wasserstein *distance* between μ and ν :

$$W_p(\mu, \nu) \triangleq \left(\inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X}^2} d(\mathbf{x}, \mathbf{y})^p \, d\pi(\mathbf{x}, \mathbf{y}) \right)^{1/p}. \quad (6.197)$$

While the symmetry and the fact that $W_p(\mu, \nu) = 0 \Rightarrow \mu = \nu$ are relatively easy to prove provided d is a metric, proving the triangle inequality is slightly more challenging, and builds on a result known as the gluing lemma ([Vil08, p.23]). The p 'th power of $W_p(\mu, \nu)$ is often abbreviated as $W_p^p(\mu, \nu)$.

6.8.3 Solving optimal transport

6.8.3.1 Duality and cost concavity

Both (6.192) and (6.196) are linear programs: their constraints and objective functions only involve summations. In that sense they admit a dual formulation (here, again, (6.199) subsumes (6.198)):

$$\max_{\mathbf{f} \in \mathbb{R}^n, \mathbf{g} \in \mathbb{R}^m, \mathbf{f} \oplus \mathbf{g} \leq C} \mathbf{f}^T \mathbf{a} + \mathbf{g}^T \mathbf{b} \quad (6.198)$$

$$\sup_{f \oplus g \leq c} \int_{\mathcal{X}} f \, d\mu + \int_{\mathcal{X}} g \, d\nu \quad (6.199)$$

where the sign \oplus denotes tensor addition for vectors, $\mathbf{f} \oplus \mathbf{g} = [\mathbf{f}_i + \mathbf{g}_j]_{ij}$, or functions, $f \oplus g : \mathbf{x}, \mathbf{y} \mapsto f(\mathbf{x}) + g(\mathbf{y})$. In other words, the dual problem looks for a pair of vectors (or functions) that attain the highest possible expectation when summed against \mathbf{a} and \mathbf{b} (or integrated against μ, ν), pending the constraint that they do not differ too much across points \mathbf{x}, \mathbf{y} , as measured by c .

The dual problems in (6.192) and (6.196) have two variables. Focusing on the continuous formulation, a closer inspection shows that it is possible, given a function f for the first measure, to compute the best possible candidate for function g . That function g should be as large as possible, yet satisfy the constraint that $g(\mathbf{y}) \leq c(\mathbf{x}, \mathbf{y}) - f(\mathbf{x})$ for all \mathbf{x}, \mathbf{y} , making

$$\forall \mathbf{y} \in \mathcal{X}, \bar{f}(\mathbf{y}) \triangleq \inf_{\mathbf{x}} c(\mathbf{x}, \mathbf{y}) - f(\mathbf{x}), \quad (6.200)$$

the optimal choice. \bar{f} is called the c -transform of f . Naturally, one may choose to start instead from g , to define an alternative c -transform:

$$\forall \mathbf{x} \in \mathcal{X}, \tilde{g}(\mathbf{x}) \triangleq \inf_{\mathbf{y}} c(\mathbf{x}, \mathbf{y}) - g(\mathbf{y}). \quad (6.201)$$

Since these transformations can only improve solutions, one may even think of applying alternatively these transformations to an arbitrary f , to define \bar{f} , \tilde{f} and so on. One can show, however, that this has little interest, since

$$\bar{\tilde{f}} = \bar{f}. \quad (6.202)$$

This remark allows, nonetheless, to narrow down the set of candidate functions to those that have already undergone such transformations. This reasoning yields the so-called set of c -concave functions, $\mathcal{F}_c \triangleq \{f \mid \exists g : \mathcal{X} \rightarrow \mathbb{R}, f = \tilde{g}\}$, which can be shown, equivalently, to be the set of functions f such that $f = \tilde{f}$. One can therefore focus our attention to c -concave functions to solve (6.199) using a so-called semi-dual formulation,

$$\sup_{f \in \mathcal{F}_c} \int_{\mathcal{X}} f \, d\mu + \int_{\mathcal{X}} \bar{f} \, d\nu. \quad (6.203)$$

Going from (6.199) to (6.203), we have removed a dual variable g and narrowed down the feasible set to \mathcal{F}_c , at the cost of introducing the highly non-linear transform \bar{f} . This reformulation is, however, very useful, in the sense that it allows us to restrict our attention to c -concave functions, notably for two important classes of cost functions c : distances and squared-Euclidean norms.

6.8.3.2 Kantorovich-Rubinstein duality and Lipschitz potentials

A striking result illustrating the interest of c -concavity is provided when c is a metric d , namely when $p = 1$ in (6.197). In that case, one can prove (exploiting notably the triangle inequality of the d) that a d -concave function f is 1-Lipschitz (one has $|f(\mathbf{x}) - f(\mathbf{y})| \leq d(\mathbf{x}, \mathbf{y})$ for any \mathbf{x}, \mathbf{y}) and such that $\bar{f} = -f$. This result translates therefore in the following identity:

$$W_1(\mu, \nu) = \sup_{f \in 1\text{-Lipschitz}} \int_{\mathcal{X}} f \, (d\mu - d\nu). \quad (6.204)$$

This result has numerous practical applications. This supremum over 1-Lipschitz functions can be efficiently approximated using wavelet coefficients of densities in low dimensions [SJ08], or heuristically in more general cases by training neural networks parameterized to be 1-Lipschitz [ACB17] using ReLU activation functions, and bounds on the entries of the weight matrices.

6.8.3.3 Monge maps as gradients of convex functions: the Brenier theorem

Another application of c -concavity lies in the case $c(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \|\mathbf{x} - \mathbf{y}\|^2$, which corresponds, up to the factor $\frac{1}{2}$, to the squared W_2 distance used between densities in an Euclidean space. The remarkable result, shown first by [Bre91], is that the Monge map solving (6.194) between two measures for that cost (taken for granted μ is regular enough, here assumed to have a density wrt the Lebesgue measure) exists and is necessarily the gradient of a convex function. In loose terms, one can show that

$$T^* = \arg \min_{T: T_\# \mu = \nu} \int_{\mathcal{X}} \frac{1}{2} \|\mathbf{x} - T(\mathbf{x})\|_2^2 \mu(d\mathbf{x}). \quad (6.205)$$

exists, and is the gradient of a convex function $u : \mathbb{R}^d \rightarrow \mathbb{R}$, namely $T^* = \nabla u$. Conversely, for any convex function u , the optimal transport map between μ and the displacement $\nabla u_\# \mu$ is necessarily equal to ∇u .

We provide a sketch of the proof: one can always exploit, for any reasonable cost function c (e.g., lower bounded and lower semi continuous), primal-dual relationships: Consider an optimal coupling P^* for (6.196), as well as an optimal c -concave dual function f^* for (6.203). This implies

in particular that $(f^*, g^* = \bar{f}^*)$ is optimal for (6.199). Complementary slackness conditions for this pair of linear programs imply that if $\mathbf{x}_0, \mathbf{y}_0$ is in the support of P^* , then necessarily (and sufficiently) $f^*(\mathbf{x}_0) + \bar{f}^*(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0)$. Suppose therefore that $\mathbf{x}_0, \mathbf{y}_0$ is indeed in the support of P^* . From the equality $f^*(\mathbf{x}_0) + \bar{f}^*(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0)$ one can trivially obtain that $\bar{f}^*(\mathbf{y}_0) = c(\mathbf{x}_0, \mathbf{y}_0) - f^*(\mathbf{x}_0)$. Yet, recall also that, by definition, $\bar{f}^*(\mathbf{y}_0) = \inf_{\mathbf{x}} c(\mathbf{x}, \mathbf{y}_0) - f^*(\mathbf{x})$. Therefore, \mathbf{x}_0 has the special property that it minimizes $\mathbf{x} \rightarrow c(\mathbf{x}, \mathbf{y}_0) - f^*(\mathbf{x})$. If, at this point, one recalls that c is assumed in this section to be $c(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2$, one has therefore that \mathbf{x}_0 verifies

$$\mathbf{x}_0 \in \operatorname{argmin}_{\mathbf{x}} \frac{1}{2}\|\mathbf{x} - \mathbf{y}_0\|^2 - f^*(\mathbf{x}). \quad (6.206)$$

Assuming f^* is differentiable, which one can prove by c -concavity, this yields the identity

$$\mathbf{y}_0 - \mathbf{x}_0 - \nabla f^*(\mathbf{x}_0) = 0 \Rightarrow \mathbf{y}_0 = \mathbf{x}_0 - \nabla f^*(\mathbf{x}_0) = \nabla \left(\frac{1}{2}\|\cdot\|^2 - f^* \right) (\mathbf{x}_0). \quad (6.207)$$

Therefore, if $(\mathbf{x}_0, \mathbf{y}_0)$ is in the support of P^* , \mathbf{y}_0 is uniquely determined, which proves P^* is in fact a Monge map “disguised” as a coupling, namely

$$P^* = (\operatorname{Id}, \nabla \left(\frac{1}{2}\|\cdot\|^2 - f^* \right))_{\sharp} \mu. \quad (6.208)$$

The end of the proof can be worked out as follows: For any function $h : \mathcal{X} \rightarrow \mathbf{R}$, one can show, using the definitions of c -transforms and the Legendre transform, that $\frac{1}{2}\|\cdot\|^2 - h$ is convex if and only if h is c -concave. An intermediate step in that proof relies on showing that $\frac{1}{2}\|\cdot\|^2 - \bar{h}$ is equal to the Legendre transform of $\frac{1}{2}\|\cdot\|^2 - h$. The function $\frac{1}{2}\|\cdot\|^2 - f^*$ above is therefore convex, by c -concavity of f^* , and the optimal transport map is itself the gradient of a convex function.

Knowing that an optimal transport map for the squared-Euclidean cost is necessarily the gradient of a convex function can prove very useful to solve (6.203). Indeed, this knowledge can be leveraged to restrict estimation to relevant families of functions, namely gradients of input-convex neural networks [AXK17], as proposed in [Mak+20] or [Kor+20], as well as arbitrary convex functions with desirable smoothness and strong-convexity constants [PdC20].

6.8.3.4 Closed forms for univariate and Gaussian distributions

Many metrics between probability distributions have closed form expressions for simple cases. The Wasserstein distance is no exception, and can be computed in close form in two important scenarios. When distributions are univariate and the cost $c(\mathbf{x}, \mathbf{y})$ is either a convex function of the difference $\mathbf{x} - \mathbf{y}$, or when $\partial c / \partial \mathbf{x} \partial \mathbf{y} < 0$ a.e., then the Wasserstein distance is essentially a comparison between the quantile functions of μ and ν . Recall that for a measure ρ , its quantile function Q_ρ is a function that takes values in $[0, 1]$ and is valued in the support of ρ , and corresponds to the (generalized) inverse map of F_ρ , the cumulative distribution function (cdf) of ρ . With these notations, one has that

$$\text{OT}_c(\mu, \nu) = \int_{[0,1]} c(Q_\mu(u), Q_\nu(u)) du \quad (6.209)$$

In particular, when c is $\mathbf{x}, \mathbf{y} \mapsto |\mathbf{x} - \mathbf{y}|$ then $\text{OT}_c(\mu, \nu)$ corresponds to the Kolmogorov-Smirnov statistic, namely the area between the cdf of μ and that of ν . If c is $\mathbf{x}, \mathbf{y} \mapsto (\mathbf{x} - \mathbf{y})^2$, we recover

simply the squared-Euclidean norm between the quantile functions of μ and ν . Note finally that the Monge map is also available in closed form, and is equal to $Q_\nu \circ F_\mu$.

The second closed form applies to so-called elliptically contoured distributions, chiefly among them Gaussian multivariate distributions [Gel90]. For two Gaussians $\mathcal{N}(\mathbf{m}_1, \Sigma_1)$ and $\mathcal{N}(\mathbf{m}_2, \Sigma_2)$ their 2-Wasserstein distance decomposes as

$$W_2^2(\mathcal{N}(\mathbf{m}_1, \Sigma_1), \mathcal{N}(\mathbf{m}_2, \Sigma_2)) = \|\mathbf{m}_1 - \mathbf{m}_2\|^2 + \mathcal{B}^2(\Sigma_1, \Sigma_2) \quad (6.210)$$

where the Bures metric \mathcal{B} reads:

$$\mathcal{B}^2(\Sigma_1, \Sigma_2) = \text{tr} \left(\Sigma_1 + \Sigma_2 - 2 \left(\Sigma_1^{\frac{1}{2}} \Sigma_2 \Sigma_1^{\frac{1}{2}} \right)^{\frac{1}{2}} \right). \quad (6.211)$$

Notice in particular that these quantities are well-defined even when the covariance matrices are not invertible, and that they collapse to the distance between means as both covariances become 0. When the first covariance matrix is invertible, one has that the optimal Monge map is given by

$$T \triangleq \mathbf{x} \mapsto A(\mathbf{x} - \mathbf{m}_1) + \mathbf{m}_2, \text{ where } A \triangleq \Sigma_1^{-\frac{1}{2}} \left(\Sigma_1^{\frac{1}{2}} \Sigma_2 \Sigma_1^{\frac{1}{2}} \right)^{\frac{1}{2}} \Sigma_1^{-\frac{1}{2}} \quad (6.212)$$

It is easy to show that T^* is indeed optimal: The fact that $T_* \mathcal{N}(\mathbf{m}_1, \Sigma_1) = \mathcal{N}(\mathbf{m}_2, \Sigma_2)$ follows from the knowledge that the affine push-forward of a Gaussian is another Gaussian. Here T is designed to push precisely the first Gaussian onto the second (and A designed to recover random variables with variance Σ_2 when starting from random variables with variance Σ_1). The optimality of T can be recovered by simply noticing that is the gradient of a convex quadratic form, since A is positive definite, and closing this proof using the Brenier theorem above.

6.8.3.5 Exact evaluation using linear program solvers

We have hinted, using duality and c -concavity, that methods based on stochastic optimization over 1-Lipschitz or convex neural networks can be employed to estimate Wasserstein distances when c is the Euclidean distance or its square. These approaches are, however, non-convex and can only reach local optima. Apart from these two cases, and the closed forms provided above, the only reliable approach to compute Wasserstein distances appears when both μ and ν are discrete measures: in that case, one can instantiate and solve the discrete (6.192) problem, or its dual (6.198) formulation. The primal problem is a canonical example of network flow problems, and can be solved with the network-simplex method in $O(nm(n+m) \log(n+m))$ complexity [AMO88], or, alternatively, with the comparable auction algorithm [BC89]. These approaches suffer from computational limitations: their cubic cost is intractable for large scale scenarios; their combinatorial flavor makes it harder to solve to parallelize simultaneously the computation of multiple optimal transport problems with a common cost matrix C .

An altogether different issue, arising from statistics, should further discourage users from using these LP formulations, notably in high-dimensional settings. Indeed, the bottleneck practitioners will most likely encounter when using (6.192) is that, in most scenarios, their goal will be to approximate the distance between two continuous measures μ, ν using only i.i.d samples contained in empirical

measures $\hat{\mu}_n, \hat{\nu}_n$. Using (6.192) to approximate the corresponding (6.196) is doomed to fail, as various results [FG15] have shown in relevant settings (notably for measures in \mathbb{R}^q) that the *sample complexity* of the estimator provided by (6.192) to approximate (6.196) is of order $1/n^{1/q}$. In other words, the gap between $W_2(\mu, \nu)$ and $W_2(\hat{\mu}_n, \hat{\nu}_n)$ is large in expectation, and decreases extremely slowly as n increases in high dimensions. Thus solving (6.196) exactly between these samples is mostly time wasted on overfitting. To address this curse of dimensionality, it is therefore extremely important in practice to approach (6.196) using a more careful strategy, one that involves regularizations that can leverage prior assumptions on μ and ν . While all approaches outlined above using neural networks can be interpreted under this light, we focus in the following on a specific approach that results in a convex problem that is relatively simple to implement, embarrassingly parallel, and with quadratic complexity.

6.8.3.6 Obtaining smoothness using entropic regularization

A computational approach to speedup the resolution of (6.192) was proposed in [Cut13], building on earlier contributions [Wil69; KY94] and a filiation to the Schrödinger bridge problem in the special case where $c = d^2$ [Léo14]. The idea rests upon regularizing the transportation cost by the Kullback-Leibler divergence of the coupling to the product measure of μ, ν ,

$$W_{c,\gamma}(\mu, \nu) \triangleq \inf_{\pi \in \Pi(\mu, \nu)} \int_{\mathcal{X}^2} d(\mathbf{x}, \mathbf{y})^p \, d\pi(\mathbf{x}, \mathbf{y}) + \gamma D_{\text{KL}}(\pi \| \mu \otimes \nu). \quad (6.213)$$

When instantiated on discrete measures, this problem is equivalent to the following γ -strongly convex problem on the set of transportation matrices (which should be compared to (6.192))

$$\text{OT}_{C,\gamma}(\mathbf{a}, \mathbf{b}) = \min_{P \in \mathbf{R}_+^{n \times m}, P\mathbf{1}_m = \mathbf{a}, P^T\mathbf{1}_n = \mathbf{b}} \langle P, C \rangle \triangleq \sum_{i,j} P_{ij} C_{ij} - \gamma \mathbb{H}(P) + \gamma (\mathbb{H}(\mathbf{a}) + \mathbb{H}(\mathbf{b})), \quad (6.214)$$

which is itself equivalent to the following dual problem (which should be compared to (6.198))

$$\text{OT}_{C,\gamma}(\mathbf{a}, \mathbf{b}) = \max_{\mathbf{f} \in \mathbb{R}^n, \mathbf{g} \in \mathbb{R}^m} \mathbf{f}^T \mathbf{a} + \mathbf{g}^T \mathbf{b} - \gamma (e^{\mathbf{f}/\gamma})^T K e^{\mathbf{g}/\gamma} + \gamma (1 + \mathbb{H}(\mathbf{a}) + \mathbb{H}(\mathbf{b})) \quad (6.215)$$

and $K \triangleq e^{-C/\gamma}$ is the elementwise exponential of $-C/\gamma$. This regularization has several benefits. Primal-dual relationships show an explicit link between the (unique) solution P_γ^* and a pair of optimal dual variables $(\mathbf{f}^*, \mathbf{g}^*)$ as

$$P_\gamma^* = \text{diag}(e^{\mathbf{f}^*/\gamma}) K \text{diag}(e^{\mathbf{g}^*/\gamma}) \quad (6.216)$$

Problem (6.215) can be solved using a fairly simple strategy that has proved very sturdy in practice: a simple block-coordinate ascent (optimizing alternatively the objective in \mathbf{f} and then \mathbf{g}), resulting in the famous Sinkhorn algorithm [Sin67], here expressed with log-sum-exp updates, starting from an arbitrary initialization for \mathbf{g} , to carry out these two updates sequentially, until they converge:

$$\mathbf{f} \leftarrow \gamma \log \mathbf{a} - \gamma \log K e^{\mathbf{g}/\gamma} \quad \mathbf{g} \leftarrow \gamma \log \mathbf{b} - \gamma \log K^T e^{\mathbf{f}/\gamma} \quad (6.217)$$

The convergence of this algorithm has been amply studied (see [CK21] and references therein). Convergence is naturally slower as γ decreases, reflecting the hardness of approaching LP solutions, as studied in [AWR17]. This regularization also has statistical benefits since, as argued in [Gen+19], the sample complexity of the regularized Wasserstein distance improves to a $O(1/\sqrt{n})$ regime, with, however, a constant in $1/\gamma^{q/2}$ that deteriorates as dimension grows.

6.9 Submodular optimization

This section is written by Jeff Bilmes.

This section provides a brief overview of submodularity in machine learning.⁶ Submodularity has an extremely simple definition. However, the “simplest things are often the most complicated to understand fully” [Sam74], and while submodularity has been studied extensively over the years, it continues to yield new and surprising insights and properties, some of which are extremely relevant to data science, machine learning, and artificial intelligence. A submodular function operates on subsets of some finite *ground set*, V . Finding a guaranteed good subset of V would ordinarily require an amount of computation exponential in the size of V . Submodular functions, however, have certain properties that make optimization either tractable or approximable where otherwise neither would be possible. The properties are quite natural, however, so submodular functions are both flexible and widely applicable to real problems. Submodularity involves an intuitive and natural diminishing returns property, stating that adding an element to a smaller set helps more than adding it to a larger set. Like convexity, submodularity allows one to efficiently find provably optimal or near-optimal solutions. In contrast to convexity, however, where little regarding maximization is guaranteed, submodular functions can be both minimized and (approximately) maximized. Submodular maximization and minimization, however, require very different algorithmic solutions and have quite different applications. It is sometimes said that submodular functions are a discrete form of convexity. This is not quite true, as submodular functions are like both convex and concave functions, but also have properties that are similar simultaneously to both convex and concave functions at the same time, but then some properties of submodularity are neither like convexity nor like concavity. Convexity and concavity, for example, can be conveyed even as univariate functions. This is impossible for submodularity, as submodular functions are defined based only on the response of the function to changes amongst different variables in a multidimensional discrete space.

6.9.1 Intuition, examples, and background

Let us define a *set function* $f : 2^V \rightarrow \mathbb{R}$ as one that assigns a value to every subset of V . The notation 2^V is the power set of V , and has size $2^{|V|}$ which means that f lives in space \mathbb{R}^{2^n} — i.e., since there are 2^n possible subsets of V , f can return 2^n distinct values. We use the notation $X + v$ as shorthand for $X \cup \{v\}$. Also, the value of an element in a given context is so widely used a concept, we have a special notation for it — the incremental value *gain* of v in the context if X is defined as $f(v|X) = f(X + v) - f(X)$. Thus, while $f(v)$ is the value of element v , $f(v|X)$ is the value of element v if you already have X . We also define the gain of set X in the context of Y as $f(X|Y) = f(X \cup Y) - f(Y)$.

6.9.1.1 Coffee, lemon, milk, and tea

As a simple example, we will explore the manner in which the value of everyday items may interact and combine, namely coffee, lemon, milk, and tea. Consider the value relationships amongst the four

6. A greatly extended version of the material in this section may be found at [Bil22].

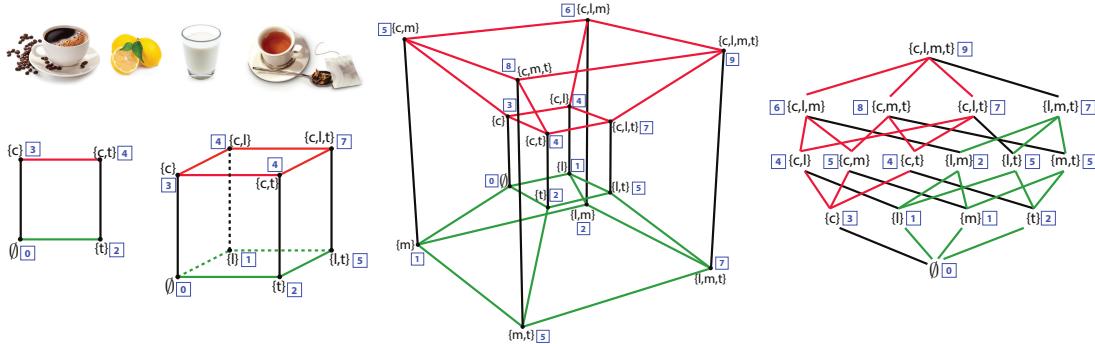


Figure 6.19: The value relationships between coffee c , lemon l , milk m , and tea t . On the left, we first see a simple square showing the relationships between coffee and tea and see that they are substitutive (or submodular). In this, and all of the shapes, the vertex label set is indicated in curly braces and the value at that vertex is a blue integer in a box. We next see a three-dimensional cube that adds lemon to the coffee and tea set. We see that tea and lemon are complementary (supermodular), but coffee and lemon are additive (modular, or independent). We next see a four-dimensional hypercube (tesseract) showing all of the value relationships described in the text. The four-dimensional hypercube is also shown as a lattice (on the right) showing the same relationships as well as two (red and green, also shown in the tesseract) of the eight three-dimensional cubes contained within.

items coffee (c), lemon (l), milk (m), and tea (t) as shown in Figure 6.19.⁷ Suppose you just woke up, and there is a function $f : 2^V \rightarrow \mathbb{R}$ that provides the average valuation for any subset of the items in V where $V = \{c, l, m, t\}$. You can think of this function as giving the average price a typical person would be willing to pay for any subset of items. Since nothing should cost nothing, we would expect that $f(\emptyset) = 0$. Clearly, one needs either coffee or tea in the morning, so $f(c) > 0$ and $f(t) > 0$, and coffee is usually more expensive than tea, so that $f(c) > f(t)$ pound for pound. Also more items cost more, so that, for example, $0 < f(c) < f(c, m) < f(c, m, t) < f(c, l, m, t)$. Thus, the function f is strictly *monotone*, or $f(X) < f(Y)$ whenever $X \subset Y$.

The next thing we note is that coffee and tea may substitute for each other — they both have the same effect, waking you up. They are mutually redundant, and they decrease each other's value since once you have had a cup of coffee, a cup of tea is less necessary and less desirable. Thus, $f(c, t) < f(c) + f(t)$, which is known as a *subadditive* relationship, the whole is less than the sum of the parts. On the other hand, some items complement each other. For example, milk and coffee are better combined together than when both are considered in isolation, or $f(m, c) > f(m) + f(c)$, a *superadditive* relationship, the whole is more than the sum of the parts. A few of the items do not affect each other's price. For example, lemon and milk cost the same together as apart, so $f(l, m) = f(l) + f(m)$, an *additive* or *modular* relationship — such a relationship is perhaps midway between a subadditive and a superadditive relationship and can be seen as a form of independence.

Things become more interesting when we consider three or more items together. For example, once you have tea, lemon becomes less valuable when you acquire milk since there might be those

7. We use different character fonts c , l , m , and t for the ingestibles than we use for other constructs. For example, below we use m for modular functions.

that prefer milk to lemon in their tea. Similarly, milk becomes less valuable once you have acquired lemon since there are those who prefer lemon in their tea to milk. So, once you have tea, lemon and milk are substitutive, you would never use both as the lemon would only curdle the milk. These are *submodular* relationships, $f(l|m, t) < f(l|t)$ and $f(m|l, t) < f(m|t)$ each of which implies that $f(l, t) + f(m, t) > f(l, m, t) + f(t)$. The value of lemon (respectively milk) with tea decreases in the larger context of having milk (respectively lemon) with tea, typical of submodular relationships.

Not all of the items are in a submodular relationship, as sometimes the presence of an item can increase the value of another item. For example, once you have milk, then tea becomes still more valuable when you also acquire lemon, since tea with the choice of either lemon or milk is more valuable than tea with the option only of milk. Similarly, once you have milk, lemon becomes more valuable when you acquire tea, since lemon with milk alone is not nearly as valuable as lemon with tea, even if milk is at hand. This means that $f(t|l, m) > f(t|m)$ and $f(l|t, m) > f(l|m)$ implying $f(l, m) + f(m, t) < f(l, m, t) + f(m)$. These are known as *supermodular* relationships, where the value increases as the context increases.

We have asked for a set of relationships amongst various subsets of the four items $V = \{c, l, m, t\}$, Is there a function that offers a value to each $X \subseteq V$ that satisfies all of the above relationships? Figure 6.19 in fact shows such a function. On the left, we see a two-dimensional square whose vertices indicate the values over subsets of $\{c, t\}$ and we can quickly verify that the sum of the blue boxes on north-west (corresponding to $f(\{c\})$) and south-east corners (corresponding to $f(\{t\})$) is greater than the sum of the north-east and south-west corners, expressing the required submodular relationship. Next on the right is a three-dimensional cube that adds the relationship with lemon. Now we have six squares, and we see that the values at each of the vertices all satisfy the above requirements — we verify this by considering the valuations at the four corners of every one of the six faces of the cube. Since $|V| = 4$, we need a four-dimensional hypercube to show all values, and this may be shown in two ways. It is first shown as a tesseract, a well-known three-dimensional projection of a four-dimensional hypercube. In the figure, all vertices are labeled both with subsets of V as well as the function value $f(X)$ as the blue number in a box. The figure on the right shows a *lattice* version of the four-dimensional hypercube, where corresponding three-dimensional cubes are shown in green and red.

We thus see that a set function is defined for all subsets of a ground set, and that they correspond to valuations at all vertices of the hypercube. For the particular function over valuations of subsets of coffee, lemon, milk, and tea, we have seen submodular, supermodular, and modular relationships all in one function. Therefore, the overall function f defined in Figure 6.19 is neither submodular, supermodular, nor modular. For combinatorial auctions, there is often a desire to have a diversity of such manners of relationships [LLN06] — representation of these relationships can be handled by a difference of submodular functions [NB05; IB12] or a sum of a submodular and supermodular function [BB18] (further described below). In machine learning, however, most of the time we are interested in functions that are submodular (or modular, or supermodular) everywhere.

6.9.2 Submodular basic definitions

For a function to be submodular, it must satisfy the submodular relationship for all subsets. We arrive at the following definition.

Definition 6.9.1 (Submodular function). *A given set function $f : 2^V \rightarrow \mathbb{R}$ is submodular if for all*

$X, Y \subseteq V$, we have the following inequality:

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y) \quad (6.218)$$

There are also many other equivalent definitions of submodularity [Bil22] some of which are more intuitive and easier to understand. For example, submodular functions are those set functions that satisfy the property of diminishing returns. If we think of a function $f(X)$ as measuring the value of a set X that is a subset of a larger set of data items $X \subseteq V$, then the submodular property means that the incremental “value” of adding a data item v to set X decreases as the size of X grows. This gives us a second classic definition of submodularity.

Definition 6.9.2 (Submodular function via diminishing returns). *A given set function $f : 2^V \rightarrow \mathbb{R}$ is submodular if for all $X, Y \subseteq V$, where $X \subseteq Y$ and for all $v \notin Y$, we have the following inequality:*

$$f(X + v) - f(X) \geq f(Y + v) - f(Y) \quad (6.219)$$

The property that the incremental value of lemon with tea is less than the incremental value of lemon once milk is already in the tea is equivalent to Equation 6.218 if we set $X = \{\text{m}, \text{t}\}$ and $Y = \{\text{l}, \text{t}\}$ (i.e., $f(\{\text{m}, \text{t}\}) + f(\{\text{l}, \text{t}\}) > f(\{\text{l}, \text{m}, \text{t}\}) + f(\{\text{t}\})$). It is naturally also equivalent to Equation 6.219 if we set $X = \{\text{t}\}$, $Y = \{\text{m}, \text{t}\}$, and with $v = \text{l}$ (i.e., $f(\{\text{l}\} \mid \{\text{m}, \text{t}\}) < f(\{\text{l}\})$).

There are many functions that are submodular, one famous one being Shannon entropy seen as a function of subsets of random variables. We first point out that there are non-negative (i.e., $f(A) \geq 0, \forall A$), monotone non-decreasing (i.e., $f(A) \leq f(B)$ whenever $A \subseteq B$) submodular functions that are not entropic [Yeu91b; ZY97; ZY98], so submodularity is not just a trivial restatement of the class of entropy functions. When a function is monotone non-decreasing, submodular, and *normalized* so that $f(\emptyset) = 0$, it is often referred to as a **polymatroid function**. Thus, while the entropy function is a polymatroid function, it does not encompass all polymatroid functions even though all polymatroid functions satisfy the properties Claude Shannon mentioned as being natural for an “information” function (see Section 6.9.7).

A function f is supermodular if and only if $-f$ is submodular. If a function is both submodular and supermodular, it is known as a *modular* function. It is always the case that a modular function $m : 2^V \rightarrow \mathbb{R}$ may take the form of a vector-scalar pair. That is, for any $A \subseteq V$, we have that $m(A) = c + \sum_{v \in A} m_v$ where c is the scalar, and $\{m_v\}_{v \in V}$ can be seen as the elements of a vector indexed by elements of V . If the modular function is normalized, so that $m(\emptyset) = 0$, then $c = 0$ and the modular function can be seen simply as a vector $m \in \mathbb{R}^V$. Hence, we sometimes say that the modular function $x \in \mathbb{R}^V$ offers a value for set A as the partial sum $x(A) = \sum_{v \in A} x(v)$. Many combinatorial problems use modular functions as objectives. For example, the graph cut problem uses a modular function defined over the edges, judges a cut in a graph as the modular function applied to the edges that comprise the cut.

As can be seen from the above, and by considering Figure 6.19, a submodular function, and in fact any set function, $f : 2^V \rightarrow \mathbb{R}$ can be seen as a function defined only on the vertices of the n -dimensional unit hypercube $[0, 1]^n$. Given any set $X \subseteq V$, we define $\mathbf{1}_X \in \{0, 1\}^V$ to be the characteristic vector of set X defined as $\mathbf{1}_X(v) = 1$ if $v \in X$ and $\mathbf{1}_X(v) = 0$ otherwise. This gives us a way to map from any set $X \subseteq V$ to a binary vector $\mathbf{1}_X$. We also see that $\mathbf{1}_X$ is itself a modular function since $\mathbf{1}_X \in \{0, 1\}^V \subset \mathbb{R}^V$.

Submodular functions share a number of properties in common with both convex and concave functions [Lov83], including wide applicability, generality, multiple representations, and closure

under a number of common operators (including mixtures, truncation, complementation, and certain convolutions). There is one important submodular closure property that we state here — that if we take non-negative weighted (or conical) combinations of submodular functions, we preserve submodularity. In other words, if we have a set of k submodular functions, $f_i : 2^V \rightarrow \mathbb{R}$, $i \in [k]$, and we form $f(X) = \sum_{i=1}^k \omega_i f_i(X)$ where $\omega_i \geq 0$ for all i , then Definition 6.9.1 immediately implies that f is also submodular. When we consider Definition 6.9.1, we see that submodular functions live in a cone in 2^n -dimensional space defined by the intersection of an exponential number of half-spaces each one of which is defined by one of the inequalities of the form $f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$. Each submodular function is therefore a point in that cone. It is therefore not surprising that taking conical combinations of such points stays within this cone.

6.9.3 Example submodular functions

As mentioned above, there are many functions that are submodular besides entropy. Perhaps the simplest such function is $f(A) = \sqrt{|A|}$ which is the composition of the square-root function (which is concave) with the cardinality $|A|$ of the set A . The gain function is $f(A + v) - f(A) = \sqrt{k+1} - \sqrt{k}$ if $|A| = k$, which we know to be a decreasing in k , thus establishing the submodularity of f . In fact, if $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is any concave function, then $f(A) = \phi(|A|)$ will be submodular for the same reason.⁸ Generalizing this slightly further, a function defined as $f(A) = \phi(\sum_{a \in A} m(a))$ is also submodular, whenever $m(a) \geq 0$ for all $a \in V$. This yields a composition of a concave function with a modular function $f(A) = \phi(m(A))$ since $\sum_{a \in A} m(a) = m(A)$. We may take sums of such functions as well as add a final modular function without losing submodularity, leading to $f(A) = \sum_{u \in U} \phi_u(\sum_{a \in A} m_u(a)) + \sum_{a \in A} m_{\pm}(a)$ where ϕ_u can be a distinct concave function for each u , $m_u(a)$ is a non-negative real value for all u and a , and $m_{\pm}(a)$ is an arbitrary real number. Therefore, $f(A) = \sum_{u \in U} \phi_u(m_u(A)) + m_{\pm}(A)$ where m_u is a u -specific non-negative modular function and m_{\pm} is an arbitrary modular function. Such functions are sometimes known as **feature-based** submodular functions [BB17] because U can be a set of non-negative features (in the machine-learning “bag-of-words” sense) and this function measures a form of dispersion over A as determined by the set of features U .

A function such as $f(A) = \sum_{u \in U} \phi_u(m_u(A))$ tends to award high diversity to a set A that has a high valuation by a distinct set of the features U . The reason is that, due to the concave nature of ϕ_u , any addition to the argument $m_u(A)$ by adding, say, v to A would diminish as A gets larger. In order to produce a set larger than A that has a much larger valuation, one must use a feature $u' \neq u$ that has not yet diminished as much.

Facility location is another well-known submodular function — perhaps an appropriate nickname would be the “ k -means of submodular functions”, due to its applicability, utility, ease-of-use (it needs only an affinity matrix), and similarity to k -medoids problems. The facility location function is defined using an affinity matrix as follows: $f(A) = \sum_{v \in V} \max_{a \in A} \text{sim}(a, v)$ where $\text{sim}(a, v)$ is a non-negative measure of the affinity (or similarity) between element a and v . Here, every element $v \in V$ must have a representative within the set A and the representative for each $v \in V$ is chosen to be the element $a \in A$ most similar to v . This function is also a form of dispersion or diversity function because, in order to maximize it, every element $v \in V$ must have some element similar to

8. While we will not be extensively discussing supermodular functions in this section, $f(A) = \phi(|A|)$ is supermodular for any convex function ϕ .

it in A . The overall score is then the sum of the similarity between each element $v \in V$ and v 's representative. This function is monotone (since as A includes more elements to become $B \supseteq A$, it is possible only to find an element in B more similar to a given v than an element in A).

While the facility location looks quite different from a feature-based function, it is possible to precisely represent any facility location function with a feature-based function. Consider just $\max_{a \in A} x_a$ and, without loss of generality, assume that $0 \leq x_1 \leq x_2 \leq \dots \leq x_n$. Then $\max_{a \in A} x_a = \sum_{i=1}^n y_i \min(|A \cap \{i, i+1, \dots, n\}|, 1)$ where $y_i = x_i - x_{i-1}$ and we set $x_0 = 0$. We note that this is a sum of weighted concave composed with modular functions since $\min(\alpha, 1)$ is concave in α , and $|A \cap \{i, i+1, \dots, n\}|$ is a modular function in A . Thus, the facility location function, a sum of these, is merely a feature-based function.

Feature-based functions, in fact, are quite expressive, and can be used to represent many different submodular functions including set cover and graph-based functions. For example, we can define a *set cover function*, given a set of sets $\{U_v\}_{v \in V}$, via $f(X) = |\bigcup_{v \in X} U_v|$. If $f(X) = |U|$ where $U = \bigcup_{v \in V} U_v$ then X indexes a set that fully covers U . This can also be represented as $f(X) = \sum_{u \in U} \min(1, m_u(X))$ where $m_u(X)$ is a modular function where $m_u(v) = 1$ if and only if $v \in U_v$ and otherwise $m_u(v) = 0$. We see that this is a feature-based submodular function since $\min(1, x)$ is concave in x , and U is a set of features.

This construct can be used to produce the vertex cover function if we set $U = V$ to be the set of vertices in a graph, and set $m_u(v) = 1$ if and only if vertices u and v are adjacent in the graph and otherwise set $m_u(v) = 0$. Similarly, the edge cover function can be expressed by setting V to be the set of edges in a graph, U to be the set of vertices in the graph, and $m_u(v) = 1$ if and only edge v is incident to vertex u .

A generalization of the set cover function is the *probabilistic coverage* function. Let $P[B_{u,v} = 1]$ be the probability of the presence of feature (or concept) u within element v . Here, we treat $B_{u,v}$ as a Bernoulli random variable for each element v and feature u so that $P[B_{u,v} = 1] = 1 - P[B_{u,v} = 0]$. Then we can define the probabilistic coverage function as $f(X) = \sum_{u \in U} f_u(X)$ where, for feature u , we have $f_u(X) = 1 - \prod_{v \in X} (1 - P[B_{u,v} = 1])$ which indicates the degree to which feature u is “covered” by X . If we set $P[B_{u,v} = 1] = 1$ if and only if $u \in U_v$ and otherwise $P[B_{u,v} = 1] = 0$, then $f_u(X) = \min(1, m_u(X))$ and the set cover function can be represented as $\sum_{u \in U} f_u(X)$. We can generalize this in two ways. First, to make it softer and more probabilistic we allow $P[B_{u,v} = 1]$ to be any number between zero and one. We also allow each feature to have a non-negative weight. This yields the general form of the probabilistic coverage function, which is defined by taking a weighted combination over all features: $f_u(X) = \sum_{u \in U} \omega_u f_u(X)$ where $\omega_u \geq 0$ is a weight for feature u . Observe that $1 - \prod_{v \in X} (1 - P[B_{u,v} = 1]) = 1 - \exp(-m_u(X)) = \phi(m_u(X))$ where m_u is a modular function with evaluation $m_u(X) = \sum_{v \in X} \log(1/(1 - P[B_{u,v} = 1]))$ and for $z \in \mathbb{R}$, $\phi(z) = 1 - \exp(-z)$ is a concave function. Thus, the probabilistic coverage function (and its set cover specialization) is also a feature-based function.

Another common submodular function is the graph cut function. Here, we measure the value of a subset of V by the edges that cross between a set of nodes and all but that set of nodes. We are given an undirected non-negative weighted graph $\mathcal{G} = (V, E, w)$ where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $w \in \mathbb{R}_+^E$ are non-negative edge weights corresponding to symmetric matrix (so $w_{i,j} = w_{j,i}$). For any $e \in E$, we have $e = \{i, j\}$ for some $i, j \in V$ with $i \neq j$, the graph cut function $f : 2^V \rightarrow \mathbb{R}$ is defined as $f(X) = \sum_{i \in X, j \in \bar{X}} w_{i,j}$ where $w_{i,j} \geq 0$ is the weight of edge $e = \{i, j\}$ ($w_{i,j} = 0$ if the edge does not exist), and where $\bar{X} = V \setminus X$ is the complement of set X . Notice that

we can write the graph cut function as follows:

$$f(X) = \sum_{i \in X, j \in \bar{X}} w_{i,j} = \sum_{i, j \in V} w_{i,j} \mathbf{1}\{i \in X, j \in \bar{X}\} \quad (6.220)$$

$$= \frac{1}{2} \sum_{i, j \in V} w_{i,j} \min(|X \cap \{i, j\}|, 1) + \frac{1}{2} \sum_{i, j \in V} w_{i,j} \min(|(V \setminus X) \cap \{i, j\}|, 1) - \frac{1}{2} \sum_{i, j \in V} w_{i,j} \quad (6.221)$$

$$= \tilde{f}(X) + \tilde{f}(V \setminus X) - \tilde{f}(V) \quad (6.222)$$

where $\tilde{f}(X) = \frac{1}{2} \sum_{i, j \in V} w_{i,j} \min(|X \cap \{i, j\}|, 1)$. Therefore, since $\min(\alpha, 1)$ is concave, and since $m_{i,j}(X) = |X \cap \{i, j\}|$ is modular, $\tilde{f}(X)$ is submodular for all i, j . Also, since $\tilde{f}(X)$ is submodular, so is $\tilde{f}(V \setminus X)$ (in X). Therefore, the graph cut function can be expressed as a sum of non-normalized feature-based functions. Note that here the second modular function is not normalized and is non-increasing, and also we subtract the constant $\tilde{f}(V)$ to achieve equality.

Another way to view the graph cut function is to consider the non-negative weights as a modular function defined over the edges. That is, we view $w \in \mathbb{R}_+^E$ as a modular function $w : 2^E \rightarrow \mathbb{R}_+$ where for every $A \subseteq E$, $w(A) = \sum_{e \in A} w(e)$ is the weight of the edges A where $w(e)$ is the weight of edge e . Then the graph cut function becomes $f(X) = w(\{(a, b) \in E : a \in X, b \in X \setminus X\})$. We view $\{(a, b) \in E : a \in X, b \in X \setminus X\}$ as a set-to-set mapping function, that maps subsets of nodes to subsets of edges, and the edge weight modular function w measures the weight of the resulting edges. This immediately suggests that other functions can measure the weight of the resulting edges as well, including non-modular functions. One example is to use a polymatroid function itself leading $h(X) = g(\{(a, b) \in E : a \in X, b \in X \setminus X\})$ where $g : 2^E \rightarrow \mathbb{R}_+$ is a submodular function defined on subsets of edges. The function h is known as the **cooperative cut** function, and it is neither submodular nor supermodular in general but there are many useful and practical algorithms that can be used to optimize it [JB16] thanks to its internal yet exposed and thus available to exploit submodular structure.

While feature-based functions are flexible and powerful, there is a strictly broader class of submodular functions, unable to be expressed by feature-based functions, that are related to deep neural networks. Here, we create a recursively nested composition of concave functions with sums of compositions of concave functions. An example is $f(A) = \phi(\sum_{u \in U} \omega_u \phi_u(\sum_{a \in A} m_u(a)))$, where ϕ is an outer concave function composed with a feature-based function, with $m_u(a) \geq 0$ and $\omega_u \geq 0$. This is known as a two-layer **deep submodular function** (DSF). A three-layer DSF has the form $f(A) = \phi(\sum_{c \in C} \omega_c \phi_c(\sum_{u \in U} \omega_{u,c} \phi_u(\sum_{a \in A} m_u(a))))$. DSFs strictly expand the class of submodular functions beyond feature-based functions, meaning that there are feature-based functions that cannot represent deep submodular functions, even simple ones [BB17].

6.9.4 Submodular optimization

Submodular functions, while discrete, would not be very useful if it was not possible to optimize over them efficiently. There are many natural problems in machine learning that can be cast as submodular optimization and that can be addressed relatively efficiently.

When one wishes to encourage diversity, information, spread, high complexity, independence, coverage, or dispersion, one usually will maximize a submodular function, in the form of $\max_{A \in \mathcal{C}} f(A)$ where $\mathcal{C} \subseteq 2^V$ is a constraint set, a set of subsets we are willing to accept as feasible solutions (more on this below).

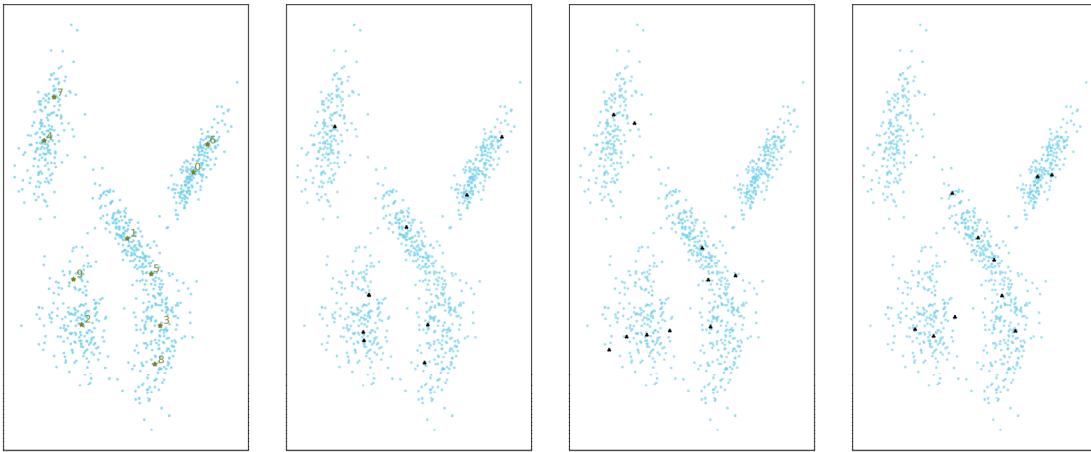


Figure 6.20: Far left: cardinality constrained (to ten) submodular maximization of a facility location function over 1000 points in two dimensions. Similarities are based on a Gaussian kernel $\text{sim}(a, v) = \exp(-d(a, v))$ where $d(\cdot, \cdot)$ is a distance. Selected points are green stars, and the greedy order is also shown next to each selected point. Right three plots: different uniformly-at-random subsets of size ten.

Why is submodularity, in general, a good model for diversity? Submodular functions are such that once you have some elements, any other elements not in your possession but that are similar to, explained by, or represented by the elements in your possession become less valuable. Thus, in order to maximize the function, one must choose other elements that are dissimilar to, or not well represented by, the ones you already have. That is, the elements similar to the ones you own are diminished in value relative to their original values, while the elements dissimilar to the ones you have do not have diminished value relative to their original values. Thus, maximizing a submodular function successfully involves choosing elements that are jointly dissimilar amongst each other, which is a definition of diversity. Diversity in general is a critically important aspect in machine learning and artificial intelligence. For example, bias in data science and machine learning can often be seen as some lack of diversity somewhere. Submodular functions have the potential to encourage (and even ensure) diversity, enhance balance, and reduce bias in artificial intelligence.

Note that in order for a submodular function to appropriately model diversity, it is important for it to be instantiated appropriately. Figure 6.20 shows an example in two dimensions. The plot compares the ten points chosen according to a facility location instantiated with a Gaussian kernel, along with the random samples of size ten. We see that the facility location selected points are more diverse and tend to cover the space much better than any of the randomly selected points, each of which miss large regions of the space and/or show cases where points near each other are jointly selected.

When one wishes for homogeneity, conformity, low complexity, coherence, or cooperation, one will usually minimize a submodular function, in the form of $\min_{A \in \mathcal{C}} f(A)$. For example, if V is a set of pixels in an image, one might wish to choose a subset of pixels corresponding to a particular object over which the properties (i.e., color, luminance, texture) are relatively homogeneous. Finding a set X of size k , even if k is large, need not have a large valuation $f(X)$, in fact it could even have the

least valuation. Thus, semantic image segmentation could work even if the object being segmented and isolated consists of the majority of image pixels.

6.9.4.1 Submodular maximization

While the cardinality constrained submodular maximization problem is NP complete [Fei98], it was shown in [NWF78; FNW78] that the very simple and efficient greedy algorithm finds an approximate solution guaranteed to be within $1 - 1/e \approx 0.63$ of the optimal solution. Moreover, the approximation ratio achieved by the simple greedy algorithm is provably the best achievable in polynomial time, assuming $P \neq NP$ [Fei98]. The greedy algorithm proceeds as follows: Starting with $X_0 = \emptyset$, we repeat the following greedy step for $i = 0 \dots (k-1)$:

$$X_{i+1} = X_i \cup (\operatorname{argmax}_{v \in V \setminus X_i} f(X_i \cup \{v\})) \quad (6.223)$$

What the above approximation result means is that if $X^* \in \operatorname{argmax}\{f(X) : |X| \leq k\}$, and if \tilde{X} is the result of the greedy procedure, then $f(\tilde{X}) \geq (1 - 1/e)f(X^*)$.

The $1 - 1/e$ guarantee is a powerful constant factor approximation result since it holds regardless of the size of the initial set V and regardless of which polymatroid function f is being optimized. It is possible to make this algorithm run extremely fast using various acceleration tricks [FNW78; NWF78; Min78].

A minor bit of additional information about a polymatroid function, however, can improve the approximation guarantee. Define the total curvature of the polymatroid function f as $\kappa = 1 - \min_{v \in V} f(v|V-v)/f(v)$ where we assume $f(v) > 0$ for all v (if not, we may prune them from the ground set since such elements can never improve a polymatroid function valuation). We thus have $0 \leq \kappa \leq 1$, and [CC84] showed that the greedy algorithm gives a guarantee of $\frac{1}{\kappa}(1 - e^{-\kappa}) \geq 1 - 1/e$. In fact, this is an equality (and we get the same bound) when $\kappa = 1$, which is the fully curved case. As κ gets smaller, the bound improves, until we reach the $\kappa = 0$ case and the bound becomes unity. Observe that $\kappa = 0$ if and only if the function is modular, in which case the greedy algorithm is optimal for the cardinality constrained maximization problem. In some cases, non-submodular functions can be decomposed into components that each might be more amenable to approximation. We see below that any set function can be written as a difference of submodular [NB05; IB12] functions, and sometimes (but not always) a given h can be composed into a monotone submodular plus a monotone supermodular function, or a BP function [BB18], i.e., $h = f + g$ where f is submodular and g is supermodular. g has an easily computed quantity called the supermodular curvature $\kappa^g = 1 - \min_{v \in V} g(v)/g(v|V-v)$ that, together with the submodular curvature, can be used to produce an approximation ratio having the form $\frac{1}{\kappa}(1 - e^{-\kappa(1-\kappa^g)})$ for greedy maximization of h .

6.9.4.2 Discrete constraints

There are many other types of constraints one might desire besides a cardinality limitation. The next simplest constraint allows each element v to have a non-negative cost, say $m(v) \in \mathbb{R}_+$. In fact, this means that the costs are modular, i.e., the cost of any set X is $m(X) = \sum_{v \in X} m(v)$. A submodular maximization problem subject to a *knapsack constraint* then takes the form $\max_{X \subseteq V : m(X) \leq b} f(X)$ where b is a non-negative budget. While the greedy algorithm does not solve this problem directly, a

slightly modified cost-scaled version of the greedy algorithm [Svi04] does solve this problem for any set of knapsack costs. This has been used for various multi-document summarization tasks [LB11; LB12].

There is no single direct analogy for a convex set when one is optimizing over subsets of the set V , but there are a few forms of discrete constraints that are both mathematically interesting and that often occur repeatedly in applications.

The first form is the independent subsets of a matroid. The independent sets of a matroid are useful to represent a constraint set for submodular maximization [Cal+07; LSV09; Lee+10], $\max_{X \in \mathcal{I}} f(X)$, and this can be useful in many ways. We can see this by showing a simple example of what is known as a *partition matroid*. Consider a partition $V = \{V_1, V_2, \dots, V_m\}$ of V into m mutually disjoint subsets that we call blocks. Suppose also that for each of the m blocks, there is a positive integer limit ℓ_i for $i \in [m]$. Consider next the set of sets formed by taking all subsets of V such that each subset has intersection with V_i no more than ℓ_i for each i . I.e., consider

$$\mathcal{I}_p = \{X : \forall i \in [m], |V_i \cap X| \leq \ell_i\}. \quad (6.224)$$

Then (V, \mathcal{I}_p) is a matroid. The corresponding submodular maximization problem is a natural generalization of the cardinality constraint in that, rather than having a fixed number of elements beyond which we are uninterested, the set of elements V is organized into groups, and here we have a fixed per-group limit beyond which we are uninterested. This is useful for fairness applications since the solution must be distributed over the blocks of the matroid. Still, there are many much more powerful types of matroids that one can use [Ox11; GM12].

Regardless of the matroid, the problem $\max_{X \in \mathcal{I}} f(X)$ can be solved, with a $1/2$ approximation factor, using the same greedy algorithm as above [NWF78; FNW78]. Indeed, the greedy algorithm has an intimate relationship with submodularity, a fact that is well studied in some of the seminal works on submodularity [Edm70; Lov83; Sch04]. It is also possible to define constraints consisting of an *intersection of matroids*, meaning that the solution must be simultaneously independent in multiple distinct matroids. Adding on to this, we might wish a set to be independent in multiple matroids and also satisfy a knapsack constraint. Knapsack constraints are not matroid constraints, since there can be multiple maximal cost solutions that are not the same size (as must be the case in a matroid). It is also possible to define discrete constraints using level sets of another completely different submodular function [IB13] — given two submodular functions f and g , this leads to optimization problems of the form $\max_{X \subseteq V: g(X) \leq \alpha} f(X)$ (the submodular cost submodular knapsack, or SCSK, problem) and $\min_{X \subseteq V: g(X) \geq \alpha} f(X)$ (the submodular cost submodular cover, or SCSC, problem). Other examples include covering constraints [IN09], and cut constraints [JB16]. Indeed, the type of constraints on submodular maximization for which good and scalable algorithms exist is quite vast, and still growing.

One last note on submodular maximization. In the above, the function f has been assumed to be a polymatroid function. There are many submodular functions that are not monotone [Buc+12]. One example we saw before, namely the graph cut function. Another example is the log of the determinant (log-determinant) of a submatrix of a positive-definite matrix (which is the Gaussian entropy plus a constant). Suppose that \mathbf{M} is an $n \times n$ symmetric positive-definite (SPD) matrix, and that \mathbf{M}_X is a row-column submatrix (i.e., it is an $|X| \times |X|$ matrix consisting of the rows and columns of \mathbf{M} consisting of the elements in X). Then the function defined as $f(X) = \log \det(\mathbf{M}_X)$ is submodular but not necessarily monotone non-decreasing. In fact, the submodularity of the log-determinant function is one of the reasons that *determinantal point processes* (DPPs), which

instantiate probability distributions over sets in such a way that high probability is given to those subsets that are diverse according to \mathbf{M} , are useful for certain tasks where we wish to probabilistically model diversity [KT11]. (See [Supplementary](#) Section 31.8.5 for details on DPPs.) Diversity of a set X here is measured by the volume of the parallelepiped which is known to be computed as the determinant of the submatrix \mathbf{M}_X and taking the log of this volume makes the function submodular in X . A DPP in fact is an example of a log-submodular probabilistic model (more in [Section 6.9.10](#)).

6.9.4.3 Submodular function minimization

In the case of a polymatroid function, unconstrained minimization is again trivial. However, even in the unconstrained case, the minimization of an arbitrary (i.e., not necessarily monotone) submodular function $\min_{X \subseteq V} f(X)$ might seem hopelessly intractable. Unconstrained submodular maximization is NP-hard (albeit approximable), and this is not surprising given that there are an exponential number of sets needing to be considered. Remarkably, submodular minimization does not require exponential computation, and is not NP-hard; in fact, there are polynomial time algorithms for doing so, something that is not at all obvious. This is one of the important characteristics that submodular functions share with convex functions, their common amenability to minimization. Starting in the very late 1960s and spearheaded by individuals such as Jack Edmonds [[Edm70](#)], there was a concerted effort in the discrete mathematics community in search of either an algorithm that could minimize a submodular function in polynomial time or a proof that such a problem was NP-hard. The nut was finally cracked in a classic paper [[GLS81](#)] on the ellipsoid algorithm that gave a polynomial time algorithm for submodular function minimization (SFM). While the algorithm was polynomial, it was a continuous algorithm, and it was not practical, so the search continued for a purely combinatorial strongly polynomial time algorithm. Queyranne [[Que98](#)] then proved that an algorithm [[NI92](#)] worked for this problem when the set function also satisfies a symmetry condition (i.e., $\forall X \subseteq V, f(X) = f(V \setminus X)$), which only requires $O(n^3)$ time. The result finally came around the year 2000 using two mostly independent methods [[IFF00](#); [Sch00](#)]. These algorithms, however, also were impractical, in that while they are polynomial time, they had unrealistically high polynomial degree (i.e., $\tilde{O}(|V|^7 * \gamma + |V|^8)$ for [[Sch00](#)] and $\tilde{O}(|V|^7 * \gamma)$ for [[IFF00](#)]). This led to additional work on combinatorial algorithms for SFM leading to algorithms that could perform SFM in time $\tilde{O}(|V|^5\gamma + |V|^6)$ in [[IO09](#)]. Two practical algorithms for SFM include the Fujishige-Wolfe procedure [[Fuj05](#); [Wol76](#)]⁹ as well as the Frank-Wolfe procedure, each of which minimize the 2-norm on a polyhedron B_f associated with the submodular function f and which is defined below (it should also be noted that the Frank-Wolfe algorithm can also be used to minimize the convex extension of the function, something that is relatively easy to compute via the Lovász extension [[Lov83](#)]). More recent work on SFM are also based on continuous relaxations of the problem in some form or another, leading algorithms with strongly polynomial running time [[LSW15](#)] of $O(|V|^3 \log^2 |V|)$ for which it was possible to drop the log factors leading to a complexity of $O(|V|^3)$ in [[Jia21](#)], weakly-polynomial running time [[LSW15](#)] of $\tilde{O}(|V|^2 \log M)$ (where $M \geq \max_{S \subseteq V} |f(S)|$), pseudopolynomial running time [[ALS20](#); [Cha+17](#)] of $\tilde{O}(|V|M^2)$, and a ϵ -approximate minimization with a linear running time [[ALS20](#)] of $\tilde{O}(|V|/\epsilon^2)$. There have been other efforts to utilize parallelism to further improve SFM [[BS20](#)].

9. This is the same Wolfe as the Wolfe in Frank-Wolfe but not the same algorithm.

6.9.5 Applications of submodularity in machine learning and AI

Submodularity arises naturally in applications in machine learning and artificial intelligence, but its utility has still not yet been as widely recognized and exploited as other techniques. For example, while information theoretic concepts like entropy and mutual information are extremely widely used in machine learning (e.g., the cross-entropy loss for classification is ubiquitous), the submodularity property of entropy is not nearly as widely explored.

Still, in the last several decades, submodularity has been increasingly studied and utilized in the context of machine learning. In the below we begin to provide only a brief survey of some of the major subareas within machine learning that have been touched by submodularity. The list is not meant to be exhaustive, or even extensive. It is hoped that the below should, at least, offer a reasonable introduction into how submodularity has been and can continue to be useful in machine learning and artificial intelligence.

6.9.6 Sketching, coresets, distillation, and data subset and feature selection

A summary is a concise representation of a body of data that can be used as an effective and efficient substitute for that data. There are many types of summaries, some being extremely simple. For example, the mean or median of a list of numbers summarizes some property (the central tendency) of that list. A random subset is also a form of summary.

Any given summary, however, is not guaranteed to do a good job serving all purposes. Moreover, a summary usually involves at least some degree of approximation and fidelity loss relative to the original, and different summaries are faithful to the original in different ways and for different tasks. For these and other reasons, the field of summarization is rich and diverse, and summarization procedures are often very specialized.

Several distinct names for summarization have been used over the past few decades, including “sketches”, “coresets”, (in the field of natural language processing) “summaries”, and “distillation”.

Sketches [Cor17; CY20; Cor+12], arose in the field of computer science and was based on the acknowledgment that data is often too large to fit in memory and too large for an algorithm to run on a given machine, something enabled by a much smaller but still representative, and provably approximate, representation of the data.

Coresets are similar to sketches and there are some properties that are more often associated with coresets than with sketches, but sometimes the distinction is a bit vague. The notion of a coreset [BHP02; AHP+05; BC08] comes from the field of computational geometry where one is interested in solving certain geometric problems based on a set of points in \mathbb{R}^d . For any geometric problem and a set of points, a coreset problem typically involves finding the smallest weighted subset of points so that when an algorithm is run on the weighted subset, it produces approximately the same answer as when it is run on the original large dataset. For example, given a set of points, one might wish to find the diameter of a set, or the radius of the smallest enclosing sphere, or finding the narrowest annulus (ring) containing the points, or a subset of points whose k -center clustering is approximately the same as the k -center clustering of the whole [BHP02].

Document summarization became one of the most important problems in natural language processing (NLP) in the 1990s although the idea of computing a summary of a text goes back much further to the 1950s [Luh58; Edm69], also and coincidentally around the same time that the CliffsNotes [Wik21] organization began. There are two main forms of document summarization [YWX17]. With *extractive*

summarization [NM12], a set of sentences (or phrases) are extracted from the documents needing to be summarized, and the resulting subset of sentences, perhaps appropriately ordered, comprises the summary.

With abstractive summarization [LN19], on the other hand, the goal is to produce an “abstract” of the documents, where one is not constrained to have any of the sentences in the abstract correspond to any of the sentences in the original documents. With abstractive summarization, therefore, the goal is to synthesize a small set of new pseudo sentences that represent the original documents. CliffsNotes, for example, are abstractive summaries of the literature being represented.

Another form of summarization that has more recently become popular in the machine learning community is *data distillation* [SG06b; Wan+20c; Suc+20; BYH20; NCL20; SS21; Ngu+21] or equivalently *dataset condensation* [ZMB21; ZB21]. With data distillation¹⁰, the goal is to produce a small set of synthetic pseudosamples that can be used, for example, to train a model. The key here is that in the reduced dataset, the samples are not compelled to be the same as, or a subset of, the original dataset.

All of the above should be contrasted with data *compression*, which in some sense is the most extreme data reduction method. With compression, either lossless or lossy, one is no longer under any obligation that the reduced form of the data must be usable, or even recognizable, by any algorithm or entity other than the decoder, or uncompression, algorithm.

6.9.6.1 Summarization Algorithm Design Choices

It is the author’s contention that the notions of summarization, coresets, sketching, and distillation are certainly analogous and quite possibly synonymous, and they are all different from compression. The different names for summarization are simply different nomenclatures for the same language game. What matters is not what you call it but the choices one makes when designing a procedure for summarization. And indeed, there are many choices.

Submodularity offers essentially an infinite number of ways to perform data sketching and coresets. When we view the submodular function as an information function (as we discussed in Section 6.9.7), where $f(X)$ is the information contained in set X and $f(V)$ is the maximum available information, finding the small X that maximizes $f(X)$ (i.e., $X^* \in \text{argmax}\{f(X) : |X| \leq k\}$), is a form of coreset computation that is parameterized by the function f which has 2^n parameters since f lives in a 2^n -dimensional cone. Performing this maximization will then minimize the residual information $f(V \setminus X|X)$ about anything not present in the summary $V \setminus X$ since $f(V) = f(X \cup V \setminus X) = f(V \setminus X|X) + f(X)$ so maximizing $f(X)$ will minimize $f(V \setminus X|X)$. For every f , moreover, the same algorithm (e.g., the greedy algorithm) can be used to produce the summarization, and in every case, there is an approximation guarantee relative to the current f , as mentioned in earlier sections, as long as f stays submodular. Hence, submodularity provides a universal framework for summarization, coresets, and sketches to the extent that the space of submodular functions itself is sufficiently diverse and spans over different coreset problems.

Overall, the coreset or sketching problem, when using submodular functions, therefore becomes a problem of “submodular design”. That is, how do we construct a submodular function that, for a particular problem, acts as a good coreset producer when the function is maximized. There are three general approaches to produce an f that works well as a summarization objective: (1) a

10. Data distillation is distinct from the notion of *knowledge distillation* [HVD14; BC14; BCNM06] or *model distillation*, where the “knowledge” contained in a large model is distilled or reduced down into a different smaller model.

pragmatic approach where the function is constructed by hand and heuristics, (2) a learning approach where all or part of the submodular function is inferred from an optimization procedure, and (3) a mathematical approach where a given submodular function when optimized offers a coresset property.

When the primary goal is a practical and scalable algorithm that can produce an extractive summary that works well on a variety of different data types, and if one is comfortable with heuristics that work well in practice, a good option is to specify a submodular function by hand. For example, given a similarity matrix, it is easy to instantiate a facility location function and maximize it to produce a summary. If there are multiple similarity matrices, one can construct multiple facility location functions and maximize their convex combination. Such an approach is viable and practical and has been used successfully many times in the past for producing good summaries. One of the earliest examples of this is the algorithm presented in [KKT03] that shows how a submodular model can be used to select the most influential nodes in a social network. Perhaps the earliest example of this approach used for data subset selection for machine learning is [LB09] which utilizes a submodular facility location function based on Fisher kernels (gradients wrt parameters of log probabilities) and applies it to unsupervised speech selection to reduce transcription costs. Other examples of this approach includes: [LB10a; LB11] which developed submodular functions for query-focused document summarization; [KB14b] which computes a subset of training data in the context of transductive learning in a statistical machine translation system; [LB10b; Wei+13; Wei+14] which develops submodular functions for speech data subset selection (the former, incidentally, is the first use of a deep submodular function and the latter does this in an unsupervised label-free fashion); [SS18a] which is a form of robust submodularity for producing coresets for training CNNs; [Kau+19] which uses a facility location to facilitate diversity selection in active learning; [Bai+15; CTN17] which develops a mixture of submodular functions for document summarization where the mixture coefficients are also included in the hyperparameter set; and [Xu+15], which uses a symmetrized submodular function for the purposes of video summarization.

The learnability and identifiability of submodular functions has received a good amount of study from a theoretical perspective. Starting with the strictest learning settings, the problem looks pretty dire. For example, [SF08; Goe+09] shows that if one is restricted to making a polynomial number of queries (i.e., training pairs of the form $(S, f(S))$) of a monotone submodular function, then it is not possible to approximate f with a multiplicative approximation factor better than $\tilde{\Omega}(\sqrt{n})$. In [BH11], goodness is judged multiplicatively, meaning for a set $A \subseteq V$ we wish that $\tilde{f}(A) \leq f(A) \leq g(n)f(A)$ for some function $g(n)$, and this is typically a probabilistic condition (i.e., measured by distribution, or $\tilde{f}(A) \leq f(A) \leq g(n)f(A)$, should happen on a fraction at least $1 - \beta$ of the points). Alternatively, goodness may also be measured by an additive approximation error, say by a norm. I.e., defining $\text{err}_p(f, \tilde{f}) = \|f - \tilde{f}\|_p = (E_{A \sim \Pr}[|f(A) - \tilde{f}(A)|^p])^{1/p}$, we may wish $\text{err}_p(f, \tilde{f}) < \epsilon$ for $p = 1$ or $p = 2$. In the PAC (probably approximately correct) model, we probably ($\delta > 0$) approximately ($\epsilon > 0$ or $g(n) > 1$) learn ($\beta = 0$) with a sample or algorithmic complexity that depends on δ and $g(n)$. In the PMAC (probably mostly approximately correct) model [BH11], we also “mostly” ($\beta > 0$) learn. In some cases, we wish to learn the best submodular approximation to a non-submodular function. In other cases, we are allowed to deviate from submodularity as long as the error is small. Learning special cases includes coverage functions [FK14; FK13a], and low-degree polynomials [FV15], curvature limited functions [IJB13], functions with a limited “goal” [DHK14; Bac+18], functions that are Fourier sparse [Wen+20a], or that are of a family called “juntas” [FV16], or that come from families other than submodular [DFF21], and still others [BRS17; FKV14; FKV17; FKV20; FKV13; YZ19]. Other results include that one cannot minimize a submodular function by learning it first

from samples [BS17]. The essential strategy of learning is to attempt to construct a submodular function approximation \hat{f} from an underlying submodular function f querying the latter only a small number of times. The overall gist of these results is that it is hard to learn everywhere and accurately.

In the machine learning community, learning can be performed extremely efficiently in practice, although there are not the types of guarantees as one finds above. For example, given a mixture of submodular components of the form $f(A) = \sum_i \alpha_i f_i(A)$, if each f_i is considered fixed, then the learning occurs only over the mixture coefficients α_i . This can be solved as a linear regression problem where the optimal coefficients can be computed in a linear regression setting. Alternatively, such functions can be learnt in a max-margin setting where the goal is primarily to adjust α_i to ensure that $f(A)$ is large on certain subsets [SSJ12; LB12; Tsc+14]. Even here there are practical challenges, however, since it is in general hard in practice to obtain a training set of pairs $\{(S_i, F(S_i))\}_i$. Alternatively, one can also “learn” a submodular function in a reinforcement learning setting [CKK17] by optimizing the implicit function directly from gain vectors queried from an environment. In general, such practical learning algorithms have been used for image summarization [Tsc+14], document summarization [LB12], and video summarization [GGG15; Vas+17a; Gon+14; SGS16; SLG17]. While none of these learning approaches claim to approximate some true underlying submodular function, in practice, they do perform better than the by-hand crafting of a submodular function mentioned above.

By a submodularity based coresets, we mean one where the direct optimization of a submodular function offers a theoretical guarantee for some specific problem. This is distinct from above where the submodular function is used as a surrogate heuristic objective function and for which, even if the submodular function is learnt, optimizing it is only a heuristic for the original problem. In some limited cases, it can be shown that the function we wish to approximate is already submodular, e.g., in the case of certain naive Bayes and k -NN classifiers [WIB15] where the training accuracy, as a function of the training data subset, can be shown to be submodular. Hence, maximizing this function offers the same guarantee on the training accuracy as it does on the submodular function. Unfortunately, the accuracy for many models is not a submodular function, although they do have a difference of submodular [NB05; IB12] decomposition.

In other cases, it can be shown that certain desirable coresets objectives are inherently submodular. For example, in [MBL20], it is shown that the normed difference between the overall gradient (from summing over all samples in the training data) and an approximate gradient (from summing over only samples in a summary) can be upper bounded with a supermodular function that, when converted to a submodular facility location function and maximized, will select a set that reduces this difference, and will lead to similar convergence rates to an approximate optimum solution in the convex case. A similar example of this in a DPP context is shown in [TBA19]. In other cases, subsets of the training data and training occur simultaneously using a continuous-discrete optimization framework, where the goal is to minimize the loss on diverse and challenging samples measured by a submodular objective [ZB18]. In still other cases, bi-level objectives related to but not guaranteed to be submodular can be formed where a set is selected from a training set with the deliberate purpose of doing well on a validation set [Kil+20; BMK20].

The methods above have focused on reducing the number of samples in a training dataset. Considering the transpose of a design matrix, however, all of the above methods can be used for reducing the features of a machine learning procedure as well. Specifically, any of the extractive summarization, subset selection, or coresets methods can be seen as feature selection while any of the abstract summarization, sketching, or distillation approaches can be seen as dimensionality

reduction.

6.9.7 Combinatorial information functions

The entropy function over a set of random variables X_1, X_2, \dots, X_n is defined as $H(X_1, X_2, \dots, X_n) = -\sum_{x_1, x_2, \dots, x_n} p(x_1, \dots, x_n) \log p(x_1, \dots, x_n)$. From this we can define three set-argument conditional mutual information functions as $I_H(A; B|C) = I(X_A; X_B|X_C)$ where the latter is the mutual information between variables indexed by A and B given variables indexed by C . This mutual information expresses the residual information between X_A and X_B that is not explained by their common information with X_C .

As mentioned above, we may view any polymatroid function as a type of information function over subsets of V . That is, $f(A)$ is the information in set A — to the extent that this is true, this property justifies f 's use as a summarization objective as mentioned above. The reason f may be viewed as an information function stems from f being normalized, f 's non-negativity, f 's monotonicity, and the property that further conditioning reduces valuation (i.e., $f(A|B) \geq f(A|B, C)$ which is identical to the submodularity property). These properties were deemed as essential to the entropy function in Shannon's original work [Sha48] but are true of any polymatroid function as well. Hence, given any polymatroid function f , is it possible to define a combinatorial mutual information function [Iye+21] in a similar way. Specifically, we can define the combinatorial (submodular) conditional mutual information (CCMI) as $I_f(A; B|C) = f(A + C) + f(B + C) - f(C) - f(A + B + C)$, which has been known as the connectivity function [Cun83] amongst other names. If f is the entropy function, then this yields the standard entropic mutual information but here the mutual information can be defined for any submodular information measure f . For an arbitrary polymatroid f , therefore, $I_f(A; B|C)$ can be seen as an A, B set-pair similarity score that ignores, neglects, or discounts any common similarity between the A, B pair that is due to C .

Historical use of a special case of CCMI, i.e., $I_f(A; B)$ where $C = \emptyset$, occurred in a number of circumstances. For example, in [GKS05] the function $g(A) = I_f(A; V \setminus A)$ (which, incidentally, is both symmetric ($g(A) = g(V \setminus A)$ for all A) and submodular) was optimized using the greedy procedure; this has a guarantee as long as $g(A)$ is monotone up $2k$ elements whenever one wishes for a summary of size k . This was done for f being the entropy function, but it can be used for any polymatroid function. In similar work, where f is the Shannon entropy function, [KG05] demonstrated that $g_C(A) = I_f(A; C)$ (for a fixed set C) is not submodular in A but if it is the case that the elements of V are independent given C then submodularity is preserved. This can be immediately seen by the consequence of this independence assumption which yields that $I_f(A; C) = f(A) - f(A|C) = f(A) - \sum_{a \in A} f(a|C)$ where the second equality is due to the conditional independence property. In this case, I_f is the difference between a submodular and a modular function which preserves submodularity for any polymatroid f .

On the other hand, it would be useful for $g_{B,C}(A) = I_f(A; B|C)$, where B and C are fixed, to be possible to optimize in terms of A . One can view this function as one that, when it is maximized, chooses A to be similar to B in a way that neglects or discounts any common similarity that A and B have with C . One option to optimize this function to utilize difference of submodular [NB05; IB12] optimization as mentioned earlier. A more recent result shows that in some cases $g_{B,C}(A)$ is still submodular in A . Define the second-order partial derivative of a submodular function f as follows $f(i, j|S) \triangleq f(j|S + i) - f(j|S)$. Then if it is the case that $f(i, j|S)$ is monotone non-decreasing in S for $S \subseteq V \setminus \{i, j\}$ then $I_f(A; B|C)$ is submodular in A for fixed B and C . It may be thought that only esoteric functions have this property, but in fact [Iye+21] shows that this is true for a number

of widely used submodular functions in practice, including the facility location function which results in the form $I_f(A; B|C) = \sum_{v \in V} \max\left(\min\left(\sum_{a \in A} \text{sim}(v, a), \max_{b \in B} \text{sim}(v, b)\right) - \max_{c \in C} \text{sim}(v, c), 0\right)$. This function was used [Kot+22] to produce summaries A that were particularly relevant to a query given by B but that should neglect information in C that can be considered “private” information to avoid.

6.9.8 Clustering, data partitioning, and parallel machine learning

There are an almost unlimited number of clustering algorithms and a plethora of reviews on their variants. Any given submodular function can also instantiate a clustering procedure as well, and there are several ways to do this. Here we offer only a brief outline of the approach. In the last section, we defined $I_f(A; V \setminus A)$ as the CCMI between A and everything but A . When we view this as a function of A , then $g(A) = I_f(A; V \setminus A)$ and $g(A)$ is a symmetric submodular function that can be minimized using Queyranne’s algorithm [Que98; NI92]. Once this is done, the resulting A is such that it is least similar to $V \setminus A$, according to $I_f(A; V \setminus A)$ and hence forms a 2-clustering. This process can then be recursively applied where we form two new functions $g_A(B) = I_f(B; A \setminus B)$ for $B \subseteq A$ and $g_{V \setminus A}(B) = I_f(B; (V \setminus A) \setminus B)$ for $B \subseteq V \setminus A$. These are two symmetric submodular functions on different ground sets that also can be minimized using Queyranne’s algorithm. This recursive bisection algorithm then repeats until the desired number of clusters is formed. Hence, the CCMI function can be used as a top-down recursive bisection clustering procedure and has been called Q-clustering [NJB05; NB06]. It should be noted that such forms of clustering often generalize forming a multiway cut in an undirected graph in which case the objective becomes the graph-cut function that, as we saw above, is also submodular. In some cases, the number of clusters need not be specified in advance [NKI10]. Another submodular approach to clustering can be found in [Wei+15b] where the goal is to minimize the maximum valued block in a partitioning which can lead to submodular load balancing or minimum makespan scheduling [HS88; LST90].

Yet another form of clustering can be seen via the simple cardinality constrained submodular maximization process itself which can be compared to a k -medoids process whenever the objective f is the facility location function. Hence, any such submodular function can be seen as a submodular-function-parameterized form of finding the k “centers” among a set of data items. There have been numerous applications of submodular clustering. For example, using these techniques it is possible to identify parcellations of the human brain [Sal+17a]. Other applications include partitioning data for more effective and accurate and lower variance distributed machine learning training [Wei+15a] and also for more ideal mini-batch construction for training deep neural networks [Wan+19b].

6.9.9 Active and semi-supervised learning

Suppose we are given dataset $\{x_i, y_i\}_{i \in V}$ consisting of $|V| = n$ samples of x, y pairs but where the labels are unknown. Samples are labeled one at a time or one mini-batch at a time, and after each labeling step t each remaining unlabeled sample is given a score $s_t(x_i)$ that indicates the potential benefit of acquiring a label for that sample. Examples include the entropy of the model’s output distribution on x_i , or a margin-based score consisting of the difference between the top and the second-from-the-top posterior probability. This produces a modular function on the unlabeled samples, $m_t(A) = \sum_{a \in A} s(x_a)$ where $A \subseteq V$. It is simple to use this modular function to produce a mini-batch active learning procedure where at each stage we form $A_t \in \operatorname{argmax}_{A \subseteq U_t: |A|=k} m_t(A)$

where U_t is the set of unlabeled samples at stage t . Then A_t is a set of size k that gets labeled, we form $U_t = U_t \setminus A_t$, update $s_t(a)$ for $a \in U_t$, and repeat. This is called **active learning**.

The reason for using active learning with mini-batches of size greater than one is that it is often inefficient to ask for a single label at a time. The problem with such a minibatch strategy, however, is that the set A_t can be redundant. The reason is that the uncertainty about every sample in A_t could be owing to the same underlying cause — even though the model is most uncertain about samples in A_t , once one sample in A_t is labeled, it may not be optimal to label the remaining samples in A_t due to this redundancy. Utilizing submodularity, therefore, can help reduce this redundancy. Suppose $f_t(A)$ is a submodular diversity model over samples at step t . At each stage, choosing the set of samples to label becomes $A_t \in \operatorname{argmax}_{A \subseteq U_t: |A|=k} m_t(A) + f_t(A)$ — A_t is selected based on a combination of both uncertainty (via $m_t(A)$) and diversity (via $f_t(A)$). This is precisely the submodular active learning approach taken in [WIB15; Kau+19].

Another quite different approach to a form of submodular “batch” active learning setting where a batch L of labeled samples are selected all at once and then used to label the rest of the unlabeled samples. This also allows the remaining unlabeled samples to be utilized in a semi-supervised framework [GB09; GB11]. In this setting, we start with a graph $G = (V, E)$ where the nodes V need to be given a binary $\{0, 1\}$ -valued label, $y \in \{0, 1\}^V$. For any $A \subseteq V$ let $y_A \in \{0, 1\}^A$ be the labels just for node set A . We also define $V(y) \subseteq V$ as $V(y) = \{v \in V : y_v = 1\}$. Hence $V(y)$ are the graph nodes labeled 1 by y and $V \setminus V(y)$ are the nodes labeled 0. Given submodular objective f , we form its symmetric CCMI variant $I_f(A) \triangleq I_f(A; V \setminus A)$ — note that $I_f(A)$ is always submodular in A . This allows $I_f(V(y))$ to determine the “smoothness” of a given candidate labeling y . For example, if I_f is the weighted graph cut function where each weight corresponds to an affinity between the corresponding two nodes, then $I_f(V(y))$ would be small if $V(y)$ (the 1-labeled nodes) do not have strong affinity with $V \setminus V(y)$ (the 0-labeled nodes). In general, however, I_f can be any symmetric submodular function. Let $L \subseteq V$ be any candidate set of nodes to be labeled, and define $\Psi(L) \triangleq \min_{T \subseteq (V \setminus L): T \neq \emptyset} I_f(T)/|T|$. Then $\Psi(L)$ measures the “strength” of L in that if $\Psi(L)$ is small, an adversary can label nodes other than L without being too unsmooth according to I_f , while if $\Psi(L)$ is large, an adversary can do no such thing. Then [GB11] showed that given a node set L to be queried, and the corresponding correct labels y_L that are completed (in a semi-supervised fashion) according to the following $y' = \operatorname{argmin}_{\hat{y} \in \{0, 1\}^V: \hat{y}_L = y_L} I_f(V(\hat{y}))$, then this results in the following bound on the true labeling $\|y - y'\|^2 \leq 2I_f(V(y))/\Psi(L)$ suggesting that we can find a good set to query by maximizing L in $\Psi(L)$, and this holds for any submodular function. Of course, it is necessary to find an underlying submodular function f that fits a given problem, and this is discussed in Section 6.9.6.

6.9.10 Probabilistic modeling

Graphical models are often used to describe factorization requirements on families of probability distributions. Factorization is not the only way, however, to describe restrictions on such families. In a graphical model, graphs describe only which random variable may directly interact with other random variables. An entirely different strategy for producing families of often-tractable probabilistic models can be produced without requiring any factorization property at all. Considering an energy function $E(x)$ where $p(x) \propto \exp(-E(x))$, factorizations correspond to there being cliques in the graph such that the graph’s tree-width often is limited. On the other hand, finding $\max_x p(x)$ is the same as finding $\min_x E(x)$, something that can be done if $E(x) = f(V(x))$ is a submodular function

(using the earlier used notation $V(x)$ to map from binary vectors to subsets of V). Even a submodular function as simple as $f(A) = \sqrt{|A|} - m(A)$ where m is modular has tree-width of $n - 1$, and this leads to an energy function $E(x)$ that allows $\max_x p(x)$ to be solved in polynomial time using submodular function minimization (see Section 6.9.4.3). Such restrictions to $E(x)$ therefore are not of the form *amongst the random variables, who is allowed to directly interact with whom*, but rather *amongst the random variables, what is the manner that they interact*. Such potential function restrictions can also combine with direct interaction restrictions as well, and this has been widely used in computer vision, leading to cases where graph-cut and graph-cut like “move making” algorithms (such as $\alpha - \beta$ swap and α -expansion algorithms) used in attractive models (see Supplementary Section 9.3.4.3). In fact, the culmination of these efforts [KZ02] lead to a rediscovery of the submodularity (or the “regular” property) as being the essential ingredient for when Markov random fields can be solved using graph cut minimization, which is a special case of submodular function minimization.

The above model can be seen as log-supermodular since $\log p(x) = -E(x) + \log 1/Z$ is a supermodular function. These are all distributions that put high probability on configurations that yield small valuation by a submodular function. Therefore, these distributions have high probability when x consists of a homogeneous set of assignments to the elements of x . For this reason, they are useful for computer vision segmentation problems (e.g., in a segment of an image, the nearby pixels should roughly be homogeneous as that is often what defines an object). The DPPs we saw above, however, are an example of a log-submodular probability distribution since $f(X) = \log \det(\mathbf{M}_X)$ is submodular. These models have high probability for diverse sets.

More generally, $E(x)$ being either a submodular or supermodular function can produce log-submodular or log-supermodular distributions, covering both cases above where the partition function takes the form $Z = \sum_{A \subseteq V} \exp(f(A))$ for objective f . Moreover, we often wish to perform tasks much more than just finding the most probable random variable assignments. This includes marginalization, computing the partition function, constrained maximization, and so on. Unfortunately, many of these more general probabilistic inference problems do not have polynomial time solutions even though the objectives are submodular or supermodular. On the other hand, such structure has opened the doors to an assortment of new probabilistic inference procedures that exploit this structure [DK14; DK15a; DTK16; ZDK15; DJK18]. Most of these methods were of the variational sort and offered bounds on the partition function Z , sometimes making use of the fact that submodular functions have easily computable semi-gradients [IB15; Fuj05] which are modular upper and lower bounds on a submodular or supermodular function that are tight at one or more subsets. Given a submodular (or supermodular) function f and a set A , it is possible to easily construct (in linear time) a modular function upper bound $m^A : 2^V \rightarrow \mathbb{R}$ and a modular function lower bound $m_A : 2^V \rightarrow \mathbb{R}$ having the properties that $m_A(X) \leq f(X) \leq m^A(X)$ for all $X \subseteq V$ and that is tight at $X = A$ meaning $m_A(A) = f(A) = m^A(A)$ [IB15]. For any modular function m , the probability function for a characteristic vector $x = \mathbf{1}_A$ becomes $p(\mathbf{1}_A) = 1/Z \exp(E(\mathbf{1}_A)) = \prod_{a \in A} \sigma(m(a)) \prod_{a \notin A} \sigma(-m(a))$ where σ is the logistic function. Thus, a modular approximation of a submodular function is like a mean-field approximation of the distribution and makes the assumption that all random variables are independent. Such an approximation can then be used to compute quantities such as upper and lower bounds on the partition function, and much else.

6.9.11 Structured norms and loss functions

Convex norms are used ubiquitously in machine learning, often as complexity penalizing regularizers (e.g., the ubiquitous p -norms for $p \geq 1$) and also sometimes as losses (e.g., squared error). Identifying new useful structured and possibly learnable sparse norms is an interesting and useful endeavor, and submodularity can help here as well. Firstly, recall the ℓ_0 or counting norm $\|x\|_0$ simply counts the number of nonzero entries in x . When we wish for a sparse solution, we may wish to regularize using $\|x\|_0$ but it both leads to an intractable combinatorial optimization problem, and it leads to an object that is not differentiable. The usual approach is to find the closest convex relaxation of this norm and that is the one norm or $\|x\|_1$. This is convex in x and has a sub-gradient structure and hence can be combined with a loss function to produce an optimizable machine learning objective, for example the lasso. On the other hand, $\|x\|_1$ has no structure, as each element of x is penalized based on its absolute value irrespective of the state of any of the other elements. There have thus been efforts to develop group norms that penalize groups or subsets of elements of x together, such as group lasso [HTW15].

It turns out that there is a way to utilize a submodular function as the regularizer. Penalizing x via $\|x\|_0$ is identical to penalizing it via $|V(x)|$ and note that $m(A) = |A|$ is a modular function. Instead, we could penalize x via $f(V(x))$ for a submodular function f . Here, any element of x being non-zero would allow for a diminishing penalty of other elements of x being zero all according to the submodular function, and such cooperative penalties can be obtained via a submodular parameterization. Like when using the zero-norm $\|x\|_0$, this leads to the same combinatorial problem due to continuous optimization of x with a penalty term of the form $f(V(x))$. To address this, we can use the Lovász extension $\check{f}(x)$ on a vector x . This function is convex, but it is not a norm, but if we consider the construct defined as $\|x\|_f = \check{f}(|x|)$, it can be shown that this satisfies all the properties of a norm for all non-trivial submodular functions [PG98; Bac+13] (i.e., those normalized submodular functions for which $f(v) > 0$ for all v). In fact, the group lasso mentioned above is a special case for a particularly simple feature-based submodular function (a sum of min-truncated cardinality functions). But in principle, the same submodular design strategies mentioned in Section 6.9.6 can be used to produce a submodular function to instantiate an appropriate convex structured norm for a given machine learning problem.

6.9.12 Conclusions

We have only barely touched the surface of submodularity and how it applies to and can benefit machine learning. For more details, see [Bil22] and the many references contained therein. Considering once again the innocuous looking submodular inequality, then very much like the definition of convexity, we observe something that belies much of its complexity while opening the gates to wide and worthwhile avenues for machine learning exploration.

PART II

Inference

7 Inference algorithms: an overview

7.1 Introduction

In the probabilistic approach to machine learning, all unknown quantities — be they predictions about the future, hidden states of a system, or parameters of a model — are treated as random variables, and endowed with probability distributions. The process of **inference** corresponds to computing the posterior distribution over these quantities, conditioning on whatever data is available.

In more detail, let $\boldsymbol{\theta}$ represent the unknown variables, and \mathcal{D} represent the known variables. Given a likelihood $p(\mathcal{D}|\boldsymbol{\theta})$ and a prior $p(\boldsymbol{\theta})$, we can compute the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ using Bayes' rule:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})}{p(\mathcal{D})} \quad (7.1)$$

The main computational bottleneck is computing the normalization constant in the denominator, which requires solving the following high dimensional integral:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \quad (7.2)$$

This is needed to convert the unnormalized joint probability of some parameter value, $p(\boldsymbol{\theta}, \mathcal{D})$, to a normalized probability, $p(\boldsymbol{\theta}|\mathcal{D})$, which takes into account all the other plausible values that $\boldsymbol{\theta}$ could have.

Once we have the posterior, we can use it to compute posterior expectations of some function of the unknown variables, i.e.,

$$\mathbb{E}[g(\boldsymbol{\theta})|\mathcal{D}] = \int g(\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (7.3)$$

By defining g in the appropriate way, we can compute many quantities of interest, such as the following:

$$\text{mean: } g(\boldsymbol{\theta}) = \boldsymbol{\theta} \quad (7.4)$$

$$\text{covariance: } g(\boldsymbol{\theta}) = (\boldsymbol{\theta} - \mathbb{E}[\boldsymbol{\theta}|\mathcal{D}])(\boldsymbol{\theta} - \mathbb{E}[\boldsymbol{\theta}|\mathcal{D}])^\top \quad (7.5)$$

$$\text{marginals: } g(\boldsymbol{\theta}) = p(\theta_1 = \theta_1^*|\boldsymbol{\theta}_{2:D}) \quad (7.6)$$

$$\text{predictive: } g(\boldsymbol{\theta}) = p(\mathbf{y}_{N+1}|\boldsymbol{\theta}) \quad (7.7)$$

$$\text{expected loss: } g(\boldsymbol{\theta}) = \ell(\boldsymbol{\theta}, a) \quad (7.8)$$

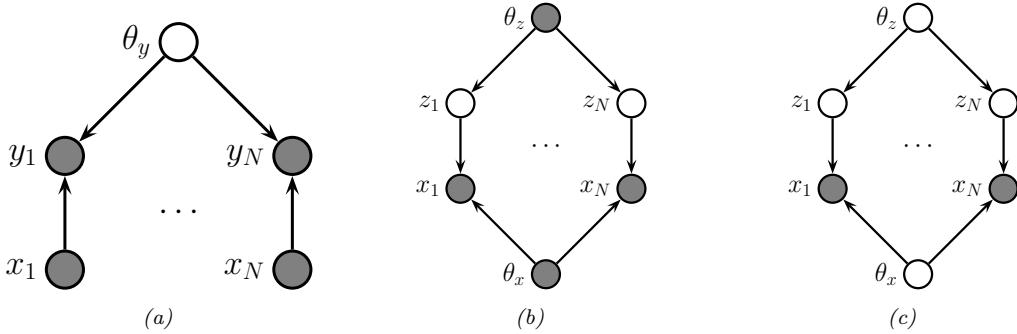


Figure 7.1: Graphical models with (a) global hidden variables for representing the Bayesian discriminative model $p(\mathbf{y}_{1:N}, \boldsymbol{\theta}_y | \mathbf{x}_{1:N}) = p(\boldsymbol{\theta}_y) \prod_{n=1}^N p(\mathbf{y}_n | \mathbf{x}_n; \boldsymbol{\theta}_y)$; (b) local hidden variables for representing the generative model $p(\mathbf{x}_{1:N}, \mathbf{z}_{1:N} | \boldsymbol{\theta}) = \prod_{n=1}^N p(\mathbf{z}_n | \boldsymbol{\theta}_z) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x)$; (c) local and global hidden variables for representing the Bayesian generative model $p(\mathbf{x}_{1:N}, \mathbf{z}_{1:N}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}_z) p(\boldsymbol{\theta}_x) \prod_{n=1}^N p(\mathbf{z}_n | \boldsymbol{\theta}_z) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x)$. Shaded nodes are assumed to be known (observed), unshaded nodes are hidden.

where \mathbf{y}_{N+1} is the next observation after seeing the N examples in \mathcal{D} , and the posterior expected loss is computing using loss function ℓ and action a (see Section 34.1.3). Finally, if we define $g(\boldsymbol{\theta}) = p(\mathcal{D} | \boldsymbol{\theta}, M)$ for model M , we can also phrase the marginal likelihood (Section 3.8.3) as an expectation wrt the prior:

$$\mathbb{E}[g(\boldsymbol{\theta}) | M] = \int g(\boldsymbol{\theta}) p(\boldsymbol{\theta} | M) d\boldsymbol{\theta} = \int p(\mathcal{D} | \boldsymbol{\theta}, M) p(\boldsymbol{\theta} | M) d\boldsymbol{\theta} = p(\mathcal{D} | M) \quad (7.9)$$

Thus we see that integration (and computing expectations) is at the heart of Bayesian inference, whereas differentiation is at the heart of optimization.

In this chapter, we give a high level summary of algorithmic techniques for computing (approximate) posteriors, and/or their corresponding expectations. We will give more details in the following chapters. Note that most of these methods are independent of the specific model. This allows problem solvers to focus on creating the best model possible for the task, and then relying on some inference engine to do the rest of the work — this latter process is sometimes called “**turning the Bayesian crank**”. For more details on Bayesian computation, see e.g., [Gel+14a; MKL21; MFR20].

7.2 Common inference patterns

There are kinds of posterior we may want to compute, but we can identify 3 main patterns, as we discuss below. These give rise to different types of inference algorithm, as we will see in later chapters.

7.2.1 Global latents

The first pattern arises when we need to perform inference in models which have **global latent variables**, such as parameters of a model $\boldsymbol{\theta}$, which are shared across all N observed training cases. This is shown in Figure 7.1a, and corresponds to the usual setting for supervised or discriminative

learning, where the joint distribution has the form

$$p(\mathbf{y}_{1:N}, \boldsymbol{\theta} | \mathbf{x}_{1:N}) = p(\boldsymbol{\theta}) \left[\prod_{n=1}^N p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}) \right] \quad (7.10)$$

The goal is to compute the posterior $p(\boldsymbol{\theta} | \mathbf{x}_{1:N}, \mathbf{y}_{1:N})$. Most of the Bayesian supervised learning models discussed in Part III follow this pattern.

7.2.2 Local latents

The second pattern arises when we need to perform inference in models which have **local latent variables**, such as hidden states $\mathbf{z}_{1:N}$; we assume the model parameters $\boldsymbol{\theta}$ are known. This is shown in Figure 7.1b. Now the joint distribution has the form

$$p(\mathbf{x}_{1:N}, \mathbf{z}_{1:N} | \boldsymbol{\theta}) = \left[\prod_{n=1}^N p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x) p(\mathbf{z}_n | \boldsymbol{\theta}_z) \right] \quad (7.11)$$

The goal is to compute $p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta})$ for each n . This is the setting we consider for most of the PGM inference methods in Chapter 9.

If the parameters are not known (which is the case for most latent variable models, such as mixture models), we may choose to estimate them by some method (e.g., maximum likelihood), and then plug in this point estimate. The advantage of this approach is that, conditional on $\boldsymbol{\theta}$, all the latent variables are conditionally independent, so we can perform inference in parallel across the data. This lets us use methods such as expectation maximization (Section 6.5.3), in which we infer $p(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta}_t)$ in the E step for all n simultaneously, and then update $\boldsymbol{\theta}_t$ in the M step. If the inference of \mathbf{z}_n cannot be done exactly, we can use variational inference, a combination known as variational EM (Section 6.5.6.1).

Alternatively, we can use a minibatch approximation to the likelihood, marginalizing out \mathbf{z}_n for each example in the minibatch to get

$$\log p(\mathcal{D}_t | \boldsymbol{\theta}_t) = \sum_{n \in \mathcal{D}_t} \log \left[\sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{z}_n | \boldsymbol{\theta}_t) \right] \quad (7.12)$$

where \mathcal{D}_t is the minibatch at step t . If the marginalization cannot be done exactly, we can use variational inference, a combination known as stochastic variational inference or SVI (Section 10.1.4). We can also learn an inference network $q_\phi(\mathbf{z} | \mathbf{x}; \boldsymbol{\theta})$ to perform the inference for us, rather than running an inference engine for each example n in each batch t ; the cost of learning $\boldsymbol{\phi}$ can be amortized across the batches. This is called amortized SVI (see Section 10.1.5).

7.2.3 Global and local latents

The third pattern arises when we need to perform inference in models which have **local and global latent variables**. This is shown in Figure 7.1c, and corresponds to the following joint distribution:

$$p(\mathbf{x}_{1:N}, \mathbf{z}_{1:N}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}_x)p(\boldsymbol{\theta}_z) \left[\prod_{n=1}^N p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}_x) p(\mathbf{z}_n | \boldsymbol{\theta}_z) \right] \quad (7.13)$$

This is essentially a Bayesian version of the latent variable model in Figure 7.1b, where now we model uncertainty in both the local variables \mathbf{z}_n and the shared global variables $\boldsymbol{\theta}$. This approach is less common in the ML community, since it is often assumed that the uncertainty in the parameters $\boldsymbol{\theta}$ is negligible compared to the uncertainty in the local variables \mathbf{z}_n . The reason for this is that the parameters are “informed” by all N data cases, whereas each local latent \mathbf{z}_n is only informed by a single datapoint, namely \mathbf{x}_n . Nevertheless, there are advantages to being “fully Bayesian”, and modeling uncertainty in both local and global variables. We will see some examples of this later in the book.

7.3 Exact inference algorithms

In some cases, we can perform example posterior inference in a tractable manner. In particular, if the prior is **conjugate** to the likelihood, the posterior will be analytically tractable. In general, this will be the case when the prior and likelihood are from the same exponential family (Section 2.4). In particular, if the unknown variables are represented by $\boldsymbol{\theta}$, then we assume

$$p(\boldsymbol{\theta}) \propto \exp(\boldsymbol{\lambda}_0^\top \mathcal{T}(\boldsymbol{\theta})) \quad (7.14)$$

$$p(\mathbf{y}_i | \boldsymbol{\theta}) \propto \exp(\tilde{\boldsymbol{\lambda}}_i(\mathbf{y}_i)^\top \mathcal{T}(\boldsymbol{\theta})) \quad (7.15)$$

where $\mathcal{T}(\boldsymbol{\theta})$ are the sufficient statistics, and $\boldsymbol{\lambda}$ are the natural parameters. We can then compute the posterior by just adding the natural parameters:

$$p(\boldsymbol{\theta} | \mathbf{y}_{1:N}) = \exp(\boldsymbol{\lambda}_*^\top \mathcal{T}(\boldsymbol{\theta})) \quad (7.16)$$

$$\boldsymbol{\lambda}_* = \boldsymbol{\lambda}_0 + \sum_{n=1}^N \tilde{\boldsymbol{\lambda}}_n(\mathbf{y}_n) \quad (7.17)$$

See Section 3.4 for details.

Another setting where we can compute the posterior exactly arises when the D unknown variables are all discrete, each with K states; in this case, the integral for the normalizing constant becomes a sum with K^D terms. In many cases, K^D will be too large to be tractable. However, if the distribution satisfies certain conditional independence properties, as expressed by a probabilistic graphical model (PGM), then we can write the joint as a product of local terms (see Chapter 4). This lets us use dynamic programming to make the computation tractable (see Chapter 9).

7.4 Approximate inference algorithms

For most probability models, we will not be able to compute marginals or posteriors exactly, so we must resort to using **approximate inference**. There are many different algorithms, which trade off speed, accuracy, simplicity, and generality. We briefly discuss some of these algorithms below, and give more detail in the following chapters. (See also [Alq22; MFR20] for a review of various methods.)

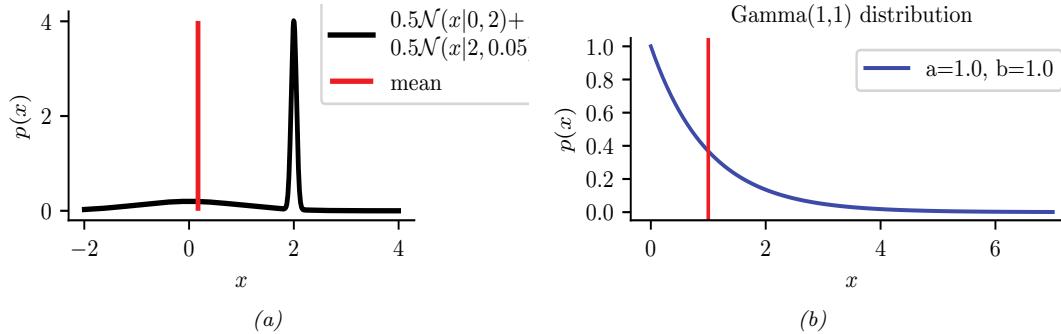


Figure 7.2: Two distributions in which the mode (highest point) is untypical of the distribution; the mean (vertical red line) is a better summary. (a) A bimodal distribution. Generated by [bimodal_dist_plot.ipynb](#). (b) A skewed $\text{Ga}(1,1)$ distribution. Generated by [gamma_dist_plot.ipynb](#).

7.4.1 The MAP approximation and its problems

The simplest approximate inference method is to compute the MAP estimate

$$\hat{\boldsymbol{\theta}} = \operatorname{argmax} p(\boldsymbol{\theta}|\mathcal{D}) = \operatorname{argmax} \log p(\boldsymbol{\theta}) + \log p(\mathcal{D}|\boldsymbol{\theta}) \quad (7.18)$$

and then to assume that the posterior puts 100% of its probability on this single value:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \quad (7.19)$$

The advantage of this approach is that we can compute the MAP estimate using a variety of optimization algorithms, which we discuss in Chapter 6. However, the MAP estimate also has various drawbacks, some of which we discuss below.

7.4.1.1 The MAP estimate gives no measure of uncertainty

In many statistical applications (especially in science) it is important to know how much one can trust a given parameter estimate. Obviously a point estimate does not convey any notion of uncertainty. Although it is possible to derive frequentist notions of uncertainty from a point estimate (see Section 3.3.1), it is arguably much more natural to just compute the posterior, from which we can derive useful quantities such as the standard error (see Section 3.2.1.6) and credible regions (see Section 3.2.1.7).

In the context of prediction (which is the main focus in machine learning), we saw in Section 3.2.2 that plugging in a point estimate can underestimate the predictive uncertainty, which can result in predictions which are not just wrong, but confidently wrong. It is generally considered very important for a predictive model to “know what it does not know”, and the Bayesian approach is a good strategy for achieving this goal.

7.4.1.2 The MAP estimate is often untypical of the posterior

In some cases, we may not be interested in uncertainty, and instead we just want a single summary of the posterior. However, the mode of a posterior distribution is often a very poor choice as a

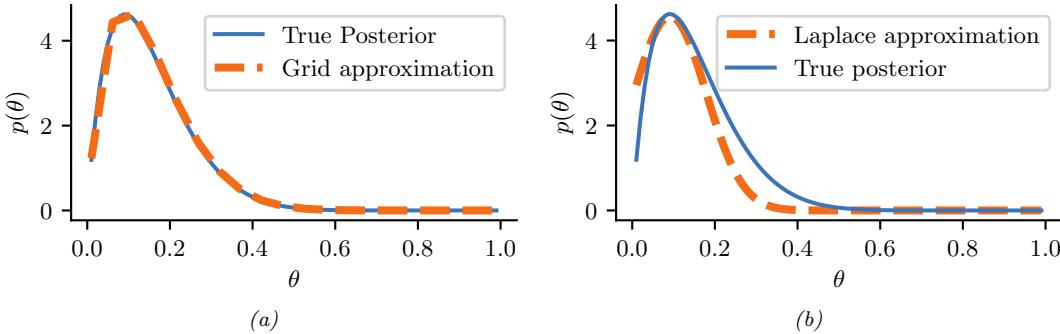


Figure 7.3: Approximating the posterior of a beta-Bernoulli model. (a) Grid approximation using 20 grid points. (b) Laplace approximation. Generated by [laplace_approx_beta_binom.ipynb](#).

summary statistic, since the mode is usually quite untypical of the distribution, unlike the mean or median. This is illustrated in Figure 7.2(a) for a 1d continuous space, where we see that the mode is an isolated peak (black line), far from most of the probability mass. By contrast, the mean (red line) is near the middle of the distribution.

Another example is shown in Figure 7.2(b): here the mode is 0, but the mean is non-zero. Such skewed distributions often arise when inferring variance parameters, especially in hierarchical models. In such cases the MAP estimate (and hence the MLE) is obviously a very bad estimate.

7.4.1.3 The MAP estimate is not invariant to reparameterization

A more subtle problem with MAP estimation is that the result we get depends on how we parameterize the probability distribution, which is not very desirable. For example, when representing a Bernoulli distribution, we should be able to parameterize it in terms of probability of success, or in terms of the log-odds (logit), without that affecting our beliefs.

For example, let $\hat{x} = \text{argmax}_x p_x(x)$ be the MAP estimate for x . Now let $y = f(x)$ be a transformation of x . In general it is not the case that $\hat{y} = \text{argmax}_y p_y(y)$ is given by $f(\hat{x})$. For example, let $x \sim \mathcal{N}(6, 1)$ and $y = f(x)$, where $f(x) = \frac{1}{1+\exp(-x+5)}$. We can use the change of variables (Section 2.5.1) to conclude $p_y(y) = p_x(f^{-1}(y)) \left| \frac{df^{-1}(y)}{dy} \right|$. Alternatively we can use a Monte Carlo approximation. The result is shown in Figure 2.12. We see that the original Gaussian for $p(x)$ has become “squashed” by the sigmoid nonlinearity. In particular, we see that the mode of the transformed distribution is not equal to the transform of the original mode.

We have seen that the MAP estimate depends on the parameterization. The MLE does not suffer from this since the likelihood is a function, not a probability density. Bayesian inference does not suffer from this problem either, since the change of measure is taken into account when integrating over the parameter space.

7.4.2 Grid approximation

If we want to capture uncertainty, we need to allow for the fact that θ may have a range of possible values, each with non-zero probability. The simplest way to capture this property is to partition

the space of possible values into a finite set of regions, call them $\mathbf{r}_1, \dots, \mathbf{r}_K$, each representing a region of parameter space of volume Δ centered on $\boldsymbol{\theta}_k$. This is called a **grid approximation**. The probability of being in each region is given by $p(\boldsymbol{\theta} \in \mathbf{r}_k | \mathcal{D}) \approx p_k \Delta$, where

$$p_k = \frac{\tilde{p}_k}{\sum_{k'=1}^K \tilde{p}_{k'}} \quad (7.20)$$

$$\tilde{p}_k = p(\mathcal{D} | \boldsymbol{\theta}_k) p(\boldsymbol{\theta}_k) \quad (7.21)$$

As K increases, we decrease the size of each grid cell. Thus the denominator is just a simple numerical approximation of the integral

$$p(\mathcal{D}) = \int p(\mathcal{D} | \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} \approx \sum_{k=1}^K \Delta \tilde{p}_k \quad (7.22)$$

As a simple example, we will use the problem of approximating the posterior of a beta-Bernoulli model. Specifically, the goal is to approximate

$$p(\boldsymbol{\theta} | \mathcal{D}) \propto \left[\prod_{n=1}^N \text{Ber}(y_n | \theta) \right] \text{Beta}(1, 1) \quad (7.23)$$

where \mathcal{D} consists of 10 heads and 1 tail (so the total number of observations is $N = 11$), with a uniform prior. Although we can compute this posterior exactly using the method discussed in Section 3.4.1, this serves as a useful pedagogical example since we can compare the approximation to the exact answer. Also, since the target distribution is just 1d, it is easy to visualize the results.

In Figure 7.3a, we illustrate the grid approximation applied to our 1d problem. We see that it is easily able to capture the skewed posterior (due to the use of an imbalanced sample of 10 heads and 1 tail). Unfortunately, this approach does not scale to problems in more than 2 or 3 dimensions, because the number of grid points grows exponentially with the number of dimensions.

7.4.3 Laplace (quadratic) approximation

In this section, we discuss a simple way to approximate the posterior using a multivariate Gaussian; this known as a **Laplace approximation** or **quadratic approximation** (see e.g., [TK86; RMC09]).

Suppose we write the posterior as follows:

$$p(\boldsymbol{\theta} | \mathcal{D}) = \frac{1}{Z} e^{-\mathcal{E}(\boldsymbol{\theta})} \quad (7.24)$$

where $\mathcal{E}(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta}, \mathcal{D})$ is called an energy function, and $Z = p(\mathcal{D})$ is the normalization constant. Performing a Taylor series expansion around the mode $\hat{\boldsymbol{\theta}}$ (i.e., the lowest energy state) we get

$$\mathcal{E}(\boldsymbol{\theta}) \approx \mathcal{E}(\hat{\boldsymbol{\theta}}) + (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^T \mathbf{g} + \frac{1}{2} (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^T \mathbf{H} (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \quad (7.25)$$

where \mathbf{g} is the gradient at the mode, and \mathbf{H} is the Hessian. Since $\hat{\boldsymbol{\theta}}$ is the mode, the gradient term is

zero. Hence

$$\hat{p}(\boldsymbol{\theta}, \mathcal{D}) = e^{-\mathcal{E}(\hat{\boldsymbol{\theta}})} \exp \left[-\frac{1}{2} (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^T \mathbf{H} (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \right] \quad (7.26)$$

$$\hat{p}(\boldsymbol{\theta} | \mathcal{D}) = \frac{1}{Z} \hat{p}(\boldsymbol{\theta}, \mathcal{D}) = \mathcal{N}(\boldsymbol{\theta} | \hat{\boldsymbol{\theta}}, \mathbf{H}^{-1}) \quad (7.27)$$

$$Z = e^{-\mathcal{E}(\hat{\boldsymbol{\theta}})} (2\pi)^{D/2} |\mathbf{H}|^{-\frac{1}{2}} \quad (7.28)$$

The last line follows from normalization constant of the multivariate Gaussian.

The Laplace approximation is easy to apply, since we can leverage existing optimization algorithms to compute the MAP estimate, and then we just have to compute the Hessian at the mode. (In high dimensional spaces, we can use a diagonal approximation.)

In Figure 7.3b, we illustrate this method applied to our 1d problem. Unfortunately we see that it is not a particularly good approximation. This is because the posterior is skewed, whereas a Gaussian is symmetric. In addition, the parameter of interest lies in the constrained interval $\theta \in [0, 1]$, whereas the Gaussian assumes an unconstrained space, $\boldsymbol{\theta} \in \mathbb{R}^D$. Fortunately, we can solve this latter problem by using a change of variable. For example, in this case we can apply the Laplace approximation to $\alpha = \text{logit}(\theta)$. This is a common trick to simplify the job of inference.

See Section 15.3.5 for an application of Laplace approximation to Bayesian logistic regression, and Section 17.3.2 for an application of Laplace approximation to Bayesian neural networks.

7.4.4 Variational inference

In Section 7.4.3, we discussed the Laplace approximation, which uses an optimization procedure to find the MAP estimate, and then approximates the curvature of the posterior at that point based on the Hessian. In this section, we discuss **variational inference (VI)**, also called **variational Bayes (VB)**. This is another optimization-based approach to posterior inference, but which has much more modeling flexibility (and thus can give a much more accurate approximation).

VI attempts to approximate an intractable probability distribution, such as $p(\boldsymbol{\theta} | \mathcal{D})$, with one that is tractable, $q(\boldsymbol{\theta})$, so as to minimize some discrepancy D between the distributions:

$$q^* = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} D(q, p) \quad (7.29)$$

where \mathcal{Q} is some tractable family of distributions (e.g., fully factorized distributions). Rather than optimizing over functions q , we typically optimize over the parameters of the function q ; we denote these **variational parameters** by ψ .

It is common to use the KL divergence (Section 5.1) as the discrepancy measure, which is given by

$$D(q, p) = D_{\text{KL}}(q(\boldsymbol{\theta} | \psi) \| p(\boldsymbol{\theta} | \mathcal{D})) = \int q(\boldsymbol{\theta} | \psi) \log \frac{q(\boldsymbol{\theta} | \psi)}{p(\boldsymbol{\theta} | \mathcal{D})} d\boldsymbol{\theta} \quad (7.30)$$

where $p(\boldsymbol{\theta} | \mathcal{D}) = p(\mathcal{D} | \boldsymbol{\theta}) p(\boldsymbol{\theta}) / p(\mathcal{D})$. The inference problem then reduces to the following optimization

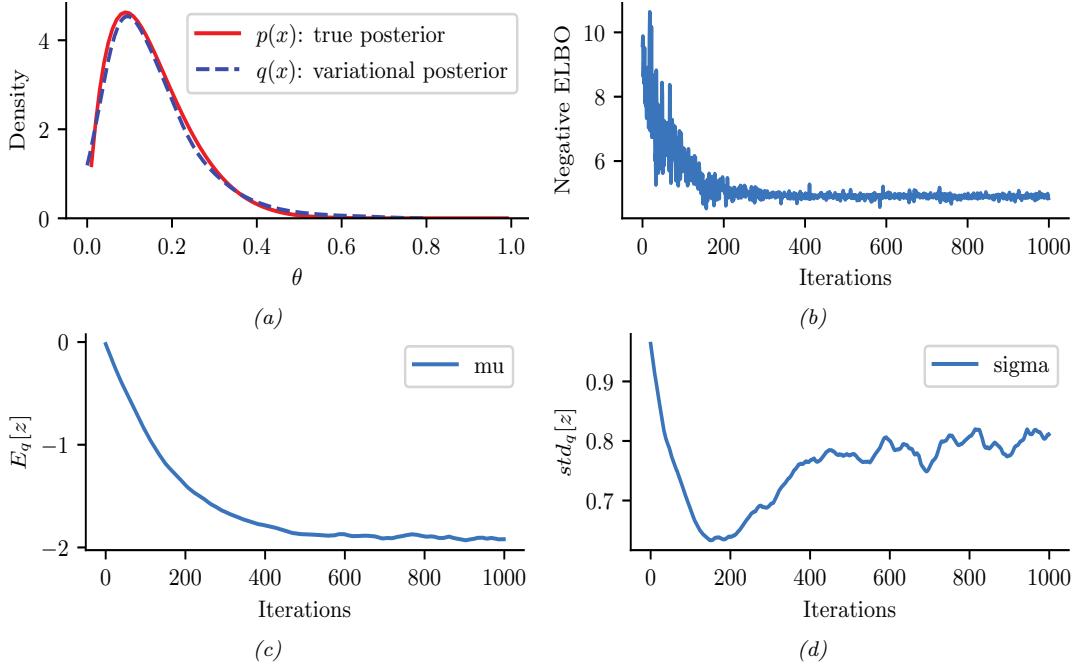


Figure 7.4: ADVI applied to the beta-Bernoulli model. (a) Approximate vs true posterior. (b) Negative ELBO over time. (c) Variational μ parameter over time. (d) Variational σ parameter over time. Generated by `advi_beta_binom.ipynb`.

problem:

$$\psi^* = \underset{\psi}{\operatorname{argmin}} D_{\text{KL}}(q(\boldsymbol{\theta}|\psi) \| p(\boldsymbol{\theta}|\mathcal{D})) \quad (7.31)$$

$$= \underset{\psi}{\operatorname{argmin}} \mathbb{E}_{q(\boldsymbol{\theta}|\psi)} \left[\log q(\boldsymbol{\theta}|\psi) - \log \left(\frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} \right) \right] \quad (7.32)$$

$$= \underset{\psi}{\operatorname{argmin}} \underbrace{\mathbb{E}_{q(\boldsymbol{\theta}|\psi)} [-\log p(\mathcal{D}|\boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) + \log q(\boldsymbol{\theta}|\psi)]}_{-\mathcal{L}(\psi)} + \log p(\mathcal{D}) \quad (7.33)$$

Note that $\log p(\mathcal{D})$ is independent of ψ , so we can ignore it when fitting the approximate posterior, and just focus on maximizing the term

$$\mathcal{L}(\psi) \triangleq \mathbb{E}_{q(\boldsymbol{\theta}|\psi)} [\log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) - \log q(\boldsymbol{\theta}|\psi)] \quad (7.34)$$

Since we have $D_{\text{KL}}(q \| p) \geq 0$, we have $\mathcal{L}(\psi) \leq \log p(\mathcal{D})$. The quantity $\log p(\mathcal{D})$, which is the log marginal likelihood, is also called the **evidence**. Hence $\mathcal{L}(\psi)$ is known as the **evidence lower bound** or **ELBO**. By maximizing this bound, we are making the variational posterior closer to the true posterior. (See Section 10.1 for details.)

We can choose any kind of approximate posterior that we like. For example, we may use a Gaussian, $q(\boldsymbol{\theta}|\psi) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. This is different from the Laplace approximation, since in VI, we optimize

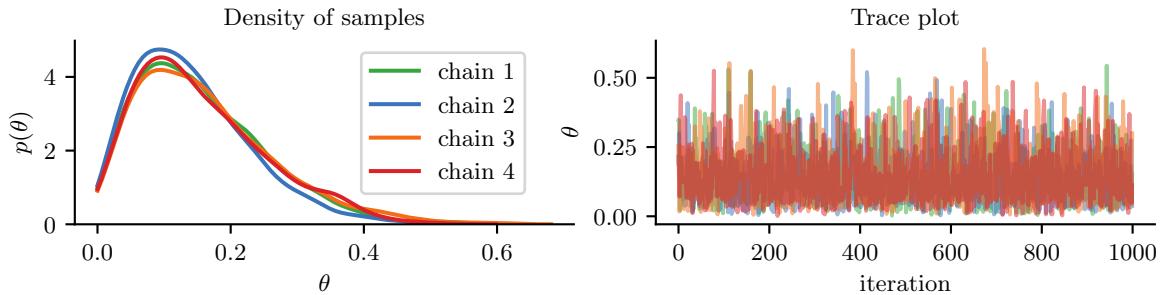


Figure 7.5: Approximating the posterior of a beta-Bernoulli model using MCMC. (a) Kernel density estimate derived from samples from 4 independent chains. (b) Trace plot of the chains as they generate posterior samples. Generated by [hmc_beta_binom.ipynb](#).

Σ , rather than equating it to the Hessian. If Σ is diagonal, we are assuming the posterior is fully factorized; this is called a **mean field** approximation.

A Gaussian approximation is not always suitable for all parameters. For example, in our 1d example we have the constraint that $\theta \in [0, 1]$. We could use a variational approximation of the form $q(\theta|\psi) = \text{Beta}(\theta|a, b)$, where $\psi = (a, b)$. However choosing a suitable form of variational distribution requires some level of expertise. To create a more easily applicable, or “turn-key”, method, that works on a wide range of models, we can use a method called **automatic differentiation variational inference** or **ADVI** [Kuc+16]. This uses the change of variables method to convert the parameters to an unconstrained form, and then computes a Gaussian variational approximation. The method also uses automatic differentiation to derive the Jacobian term needed to compute the density of the transformed variables. See Section 10.2.2 for details.

We now apply ADVI to our 1d beta-Bernoulli model. Let $\theta = \sigma(z)$, where we replace $p(\theta|\mathcal{D})$ with $q(z|\psi) = \mathcal{N}(z|\mu, \sigma)$, where $\psi = (\mu, \sigma)$. We optimize a stochastic approximation to the ELBO using SGD. The results are shown in Figure 7.4 and seem reasonable.

7.4.5 Markov chain Monte Carlo (MCMC)

Although VI is fast, it can give a biased approximation to the posterior, since it is restricted to a specific function form $q \in \mathcal{Q}$. A more flexible approach is to use a non-parametric approximation in terms of a set of samples, $q(\theta) \approx \frac{1}{S} \sum_{s=1}^S \delta(\theta - \theta^s)$. This is called a **Monte Carlo approximation**. The key issue is how to create the posterior samples $\theta^s \sim p(\theta|\mathcal{D})$ efficiently, without having to evaluate the normalization constant $p(\mathcal{D}) = \int p(\theta, \mathcal{D}) d\theta$.

For low dimensional problems, we can use methods such as **importance sampling**, which we discuss in Section 11.5. However, for high dimensional problems, it is more common to use **Markov chain Monte Carlo** or **MCMC**. We give the details in Chapter 12, but give a brief introduction here.

The most common kind of MCMC is known as the **Metropolis-Hastings algorithm**. The basic idea behind MH is as follows: we start at a random point in parameter space, and then perform a random walk, by sampling new states (parameters) from a **proposal distribution** $q(\theta'|\theta)$. If q is chosen carefully, the resulting Markov chain distribution will satisfy the property that the fraction of

time we visit each point in space is proportional to the posterior probability. The key point is that to decide whether to move to a newly proposed point $\boldsymbol{\theta}'$ or to stay in the current point $\boldsymbol{\theta}$, we only need to evaluate the unnormalized density ratio

$$\frac{p(\boldsymbol{\theta}|\mathcal{D})}{p(\boldsymbol{\theta}'|\mathcal{D})} = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})/p(\mathcal{D})}{p(\mathcal{D}|\boldsymbol{\theta}')p(\boldsymbol{\theta}')/p(\mathcal{D})} = \frac{p(\mathcal{D}, \boldsymbol{\theta})}{p(\mathcal{D}, \boldsymbol{\theta}')} \quad (7.35)$$

This avoids the need to compute the normalization constant $p(\mathcal{D})$. (In practice we usually work with log probabilities, instead of joint probabilities, to avoid numerical issues.)

We see that the input to the algorithm is just a function that computes the log joint density, $\log p(\boldsymbol{\theta}, \mathcal{D})$, as well as a proposal distribution $q(\boldsymbol{\theta}'|\boldsymbol{\theta})$ for deciding which states to visit next. It is common to use a Gaussian distribution for the proposal, $q(\boldsymbol{\theta}'|\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}'|\boldsymbol{\theta}, \sigma\mathbf{I})$; this is called the **random walk Metropolis** algorithm. However, this can be very inefficient, since it is blindly walking through the space, in the hopes of finding higher probability regions.

In models that have conditional independence structure, it is often easy to compute the **full conditionals** $p(\boldsymbol{\theta}_d|\boldsymbol{\theta}_{-d}, \mathcal{D})$ for each variable d , one at a time, and then sample from them. This is like a stochastic analog of coordinate ascent, and is called **Gibbs sampling** (see Section 12.3 for details).

For models where all unknown variables are continuous, we can often compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}, \mathcal{D})$. We can use this gradient information to guide the proposals into regions of space with higher probability. This approach is called **Hamiltonian Monte Carlo** or **HMC**, and is one of the most widely used MCMC algorithms due to its speed. For details, see Section 12.5.

We apply HMC to our beta-Bernoulli model in Figure 7.5. (We use a logit transformation for the parameter.) In panel b, we show samples generated by the algorithm from 4 parallel Markov chains. We see that they oscillate around the true posterior, as desired. In panel a, we compute a kernel density estimate from the posterior samples from each chain; we see that the result is a good approximation to the true posterior in Figure 7.3.

7.4.6 Sequential Monte Carlo

MCMC is like a stochastic local search algorithm, in that it makes moves through the state space of the posterior distribution, comparing the current value to proposed neighboring values. An alternative approach is to use perform inference using a sequence of different distributions, from simpler to more complex, with the final distribution being equal to the target posterior. This is called **sequential Monte Carlo** or **SMC**. This approach, which is more similar to tree search than local search, has various advantages over MCMC, which we discuss in Chapter 13.

A common application of SMC is to **sequential Bayesian inference**, in which we recursively compute (i.e., in an online fashion) the posterior $p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t})$, where $\mathcal{D}_{1:t} = \{(\mathbf{x}_n, y_n) : n = 1 : t\}$ is all the data we have seen so far. This sequence of distributions converges to the full batch posterior $p(\boldsymbol{\theta}|\mathcal{D})$ once all the data has been seen. However, the approach can also be used when the data is arriving in a continual, unending stream, as in state-space models (see Chapter 29). The application of SMC to such dynamical models is known as **particle filtering**. See Section 13.2 for details.

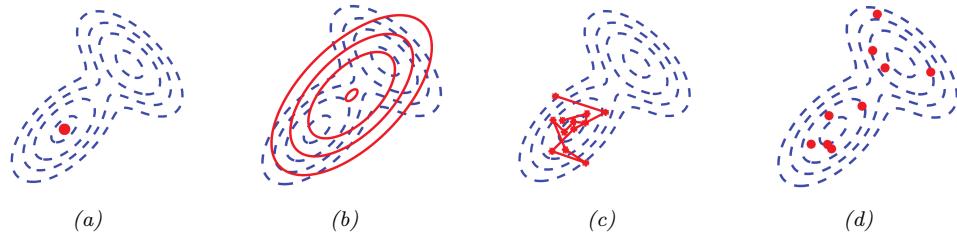


Figure 7.6: Different approximations to a bimodal 2d distribution. (a) Local MAP estimate. (b) Parametric Gaussian approximation. (c) Correlated samples from near one mode. (d) Independent samples from the distribution. Adapted from Figure 2 of [PY14]. Used with kind permission of George Panadreou.

7.4.7 Challenging posteriors

In many applications, the posterior can be high dimensional and multimodal. Approximating such distributions can be quite challenging. In Figure 7.6, we give a simple 2d example. We compare MAP estimation (which does not capture any uncertainty), a Gaussian parametric approximation such as the Laplace approximation or variational inference (see panel b), and a nonparametric approximation in terms of samples. If the samples are generated from MCMC, they are serially correlated, and may only explore a local model (see panel c). However, ideally we can draw independent samples from the entire support of the distribution, as shown in panel d. We may also be able to fit a local parametric approximation around each such sample (see Section 17.3.9.1), to get a semi-parametric approximation to the posterior.

7.5 Evaluating approximate inference algorithms

There are many different approximate inference algorithms, each of which make different tradeoffs between speed, accuracy, generality, simplicity, etc. This makes it hard to compare them on an equal footing.

One approach is to evaluate the accuracy of the approximation $q(\boldsymbol{\theta})$ by comparing to the “true” posterior $p(\boldsymbol{\theta}|\mathcal{D})$, computed offline with an “exact” method. We are usually interested in accuracy vs speed tradeoffs, which we can compute by evaluating $D_{\text{KL}}(p(\boldsymbol{\theta}|\mathcal{D}) \parallel q_t(\boldsymbol{\theta}))$, where $q_t(\boldsymbol{\theta})$ is the approximate posterior after t units of compute time. Of course, we could use other measures of distributional similarity, such as Wasserstein distance.

Unfortunately, it is usually impossible to compute the true posterior $p(\boldsymbol{\theta}|\mathcal{D})$. A simple alternative is to evaluate the quality in terms of its prediction abilities on out of sample observed data, similar to cross validation. More generally, we can compare the expected loss or Bayesian risk (Section 34.1.3) of different posteriors, as proposed in [KPS98; KPS99]:

$$R = \mathbb{E}_{p^*(\mathbf{x}, \mathbf{y})} [\ell(\mathbf{y}, q(\mathbf{y}|\mathbf{x}, \mathcal{D}))] \quad \text{where } q(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) q(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta} \quad (7.36)$$

where $\ell(\mathbf{y}, q(\mathbf{y}))$ is some loss function, such as log-loss. Alternatively, we can measure performance of the posterior when it is used in some downstream task, such as continual or active learning, as proposed in [Far22].

For some specialized methods for assessing variational inference, see [Yao+18b; Hug+20], and for Monte Carlo methods, see [CGR06; CTM17; GAR16].

8 Gaussian filtering and smoothing

8.1 Introduction

In this chapter, we consider the task of posterior inference in **state-space models** (SSMs). We discuss SSMs in more detail in Chapter 29, but we can think of them as latent variable sequence models with the conditional independencies shown by the chain-structured graphical model Figure 8.1. The corresponding joint distribution has the form

$$p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T} | \mathbf{u}_{1:T}) = \left[p(\mathbf{z}_1 | \mathbf{u}_1) \prod_{t=2}^T p(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{u}_t) \right] \left[\prod_{t=1}^T p(\mathbf{y}_t | \mathbf{z}_t, \mathbf{u}_t) \right] \quad (8.1)$$

where \mathbf{z}_t are the hidden variables at time t , \mathbf{y}_t are the observations (outputs), and \mathbf{u}_t are the optional inputs. The term $p(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{u}_t)$ is called the **dynamics model** or **transition model**, $p(\mathbf{y}_t | \mathbf{z}_t, \mathbf{u}_t)$ is called the **observation model** or **measurement model**, and $p(\mathbf{z}_1 | \mathbf{u}_1)$ is the prior or initial state distribution.¹

8.1.1 Inferential goals

Given the sequence of observations, and a known model, one of the main tasks with SSMs is to perform posterior inference about the hidden states; this is also called **state estimation**.

For example, consider an airplane flying in the sky. (For simplicity, we assume the world is 2d, not 3d.) We would like to estimate its location and velocity $\mathbf{z}_t \in \mathbb{R}^4$ given noisy sensor measurements of its location $\mathbf{y}_t \in \mathbb{R}^2$, as illustrated in Figure 8.2(a). (We ignore the inputs \mathbf{u}_t for simplicity.)

We discuss a suitable SSM for this problem, that embodies Newton's laws of motion, in Section 8.2.1.1. We can use the model to compute the **belief state** $p(\mathbf{z}_t | \mathbf{y}_{1:t})$; this is called **Bayesian filtering**. If we represent the belief state using a Gaussian, then we can use the **Kalman filter** to solve this task, as we discuss in Section 8.2.2. In Figure 8.2(b) we show the results of this algorithm. The green dots are the noisy observations, the red line shows the posterior mean estimate of the location, and the black circles show the posterior covariance. (The posterior over the velocity is not shown.) We see that the estimated trajectory is less noisy than the raw data, since it incorporates prior knowledge about how the data was generated.

Another task of interest is the **smoothing** problem where we want to compute $p(\mathbf{z}_t | \mathbf{y}_{1:T})$ using an offline dataset. We can compute these quantities using the **Kalman smoother** described in

1. In some cases, the initial state distribution is denoted by $p(\mathbf{z}_0)$, and then we derive $p(\mathbf{z}_1 | \mathbf{u}_1)$ by passing $p(\mathbf{z}_0)$ through the dynamics model. In this case, the joint distribution represents $p(\mathbf{y}_{1:T}, \mathbf{z}_{0:T} | \mathbf{u}_{1:T})$.

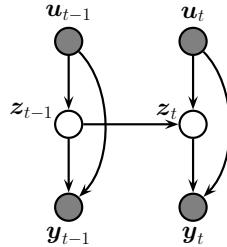


Figure 8.1: A state-space model represented as a graphical model. z_t are the hidden variables at time t , y_t are the observations (outputs), and u_t are the optional inputs.

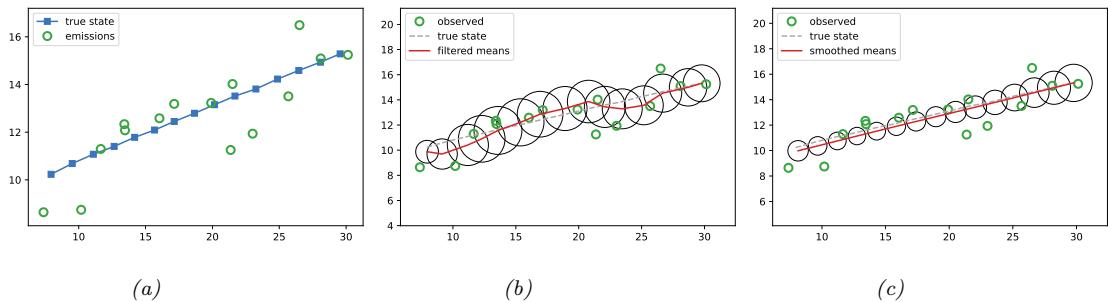


Figure 8.2: Illustration of Kalman filtering and smoothing for a linear dynamical system. (a) Observations (green circles) are generated by an object moving to the right (true location denoted by blue squares). (b) Results of online Kalman filtering. Circles are 95% confidence ellipses, whose center is the posterior mean, and whose shape is derived from the posterior covariance. (c) Same as (b), but using offline Kalman smoothing. The MSE in the trajectory for filtering is 3.13, and for smoothing is 1.71. Generated by [kf_tracking_script.ipynb](#).

Section 8.2.3. In Figure 8.2(c) we show the result of this algorithm. We see that the resulting estimate is smoother compared to filtering, and that the posterior uncertainty is reduced (as visualized by the smaller confidence ellipses).

To understand this behavior intuitively, consider a detective trying to figure out who committed a crime. As they move through the crime scene, their uncertainty is high until he finds the key clue; then they have an “aha” moment, the uncertainty is reduced, and all the previously confusing observations are, in **hindsight**, easy to explain. Thus we see that, given all the data (including finding the clue), it is much easier to infer the state of the world.

A disadvantage of the smoothing method is that we have to wait until all the data has been observed before we start performing inference, so it cannot be used for online or realtime problems. **Fixed lag smoothing** is a useful compromise between online and offline estimation; it involves computing $p(z_{t-\ell}|y_{1:t})$, where $\ell > 0$ is called the lag. This gives better performance than filtering, but incurs a slight delay. By changing the size of the lag, we can trade off accuracy vs delay. See Figure 8.3 for an illustration.

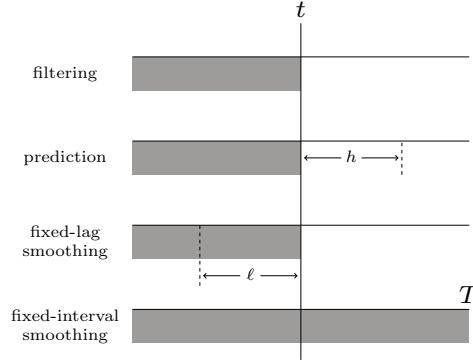


Figure 8.3: The main kinds of inference for state-space models. The shaded region is the interval for which we have data. The arrow represents the time step at which we want to perform inference. t is the current time, T is the sequence length, ℓ is the lag, and h is the prediction horizon. Used with kind permission of Peter Chang.

In addition to inferring the latent state, we may want to predict future observations. We can compute the **observed predictive distribution** h steps into the future as follows:

$$p(\mathbf{y}_{t+h} | \mathbf{y}_{1:t}) = \sum_{\mathbf{z}_{t+h}} p(\mathbf{y}_{t+h} | \mathbf{z}_{t+h}) p(\mathbf{z}_{t+h} | \mathbf{y}_{1:t}) \quad (8.2)$$

where the **hidden state predictive distribution** is obtained by pushing the current belief state through the dynamics model

$$p(\mathbf{z}_{t+h} | \mathbf{y}_{1:t}) = \sum_{\mathbf{z}_{t:t+h-1}} p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_{t+2} | \mathbf{z}_{t+1}) \cdots p(\mathbf{z}_{t+h} | \mathbf{z}_{t+h-1}) \quad (8.3)$$

(When the states are continuous, we need to replace the sums with integrals.)

8.1.2 Bayesian filtering equations

The **Bayes filter** is an algorithm for recursively computing the **belief state** $p(\mathbf{z}_t | \mathbf{y}_{1:t})$ given the prior belief from the previous step, $p(\mathbf{z}_{t-1} | \mathbf{y}_{1:t-1})$, the new observation \mathbf{y}_t , and the model. This can be done using **sequential Bayesian updating**, and requires a constant amount of computation per time step (independent of t). For a dynamical model, this reduces to the **predict-update** cycle described below.

The **prediction step** is just the **Chapman-Kolmogorov equation**:

$$p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) = \int p(\mathbf{z}_t | \mathbf{z}_{t-1}) p(\mathbf{z}_{t-1} | \mathbf{y}_{1:t-1}) d\mathbf{z}_{t-1} \quad (8.4)$$

The prediction step computes the one-step-ahead predictive distribution for the latent state, which

updates the posterior from the previous time step into the prior for the current step.²

The **update step** is just Bayes' rule:

$$p(\mathbf{z}_t | \mathbf{y}_{1:t}) = \frac{1}{Z_t} p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) \quad (8.5)$$

where the normalization constant is

$$Z_t = \int p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) d\mathbf{z}_t = p(\mathbf{y}_t | \mathbf{y}_{1:t-1}) \quad (8.6)$$

We can use the normalization constants to compute the log likelihood of the sequence as follows:

$$\log p(\mathbf{y}_{1:T}) = \sum_{t=1}^T \log p(\mathbf{y}_t | \mathbf{y}_{1:t-1}) = \sum_{t=1}^T \log Z_t \quad (8.7)$$

where we define $p(\mathbf{y}_1 | \mathbf{y}_0) = p(\mathbf{y}_1)$. This quantity is useful for computing the MLE of the parameters.

8.1.3 Bayesian smoothing equations

In the offline setting, we want to compute $p(\mathbf{z}_t | \mathbf{y}_{1:T})$, which is the belief about the hidden state at time t given all the data, both past and future. This is called (fixed interval) **smoothing**. We first perform the forwards or filtering pass, and then compute the smoothed belief states by working backwards, from right (time $t = T$) to left ($t = 1$), as we explain below. Hence this method is also called **forwards filtering backwards smoothing** or **FFBS**.

Suppose, by induction, that we have already computed $p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})$. We can convert this into a joint smoothed distribution over two consecutive time steps using

$$p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T}) \quad (8.8)$$

To derive the first term, note that from the Markov properties of the model, and Bayes' rule, we have

$$p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:t}, \mathbf{y}_{t+1:T}) \quad (8.9)$$

$$= \frac{p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} \quad (8.10)$$

$$= \frac{p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} \quad (8.11)$$

Thus the joint distribution over two consecutive time steps is given by

$$p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T}) = \frac{p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} \quad (8.12)$$

2. The prediction step is not needed at $t = 1$ if $p(\mathbf{z}_1)$ is provided as input to the model. However, if we just provide $p(\mathbf{z}_0)$, we need to compute $p(\mathbf{z}_1 | \mathbf{y}_{1:0}) = p(\mathbf{z}_1)$ by applying the prediction step.

from which we get the new smoothed marginal distribution:

$$p(\mathbf{z}_t | \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{y}_{1:t}) \int \left[\frac{p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} \right] d\mathbf{z}_{t+1} \quad (8.13)$$

$$= \int p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}) \frac{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} d\mathbf{z}_{t+1} \quad (8.14)$$

Intuitively we can interpret this as follows: we start with the two-slice filtered distribution, $p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t})$, and then we divide out the old $p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})$ and multiply in the new $p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})$, and then marginalize out \mathbf{z}_{t+1} .

8.1.4 The Gaussian ansatz

In general, computing the integrals required to implement Bayesian filtering and smoothing is intractable. However, there are two notable exceptions: if the state space is discrete, as in an HMM, we can represent the belief states as discrete distributions (histograms), which we can update using the forwards-backwards algorithm, as discussed in Section 9.2; and if the SSM is a linear-Gaussian model, then we can represent the belief states by Gaussians, which we can update using the Kalman filter and smoother, which we discuss in Section 8.2.2 and Section 8.2.3. In the nonlinear and/or non-Gaussian setting, we can still use a Gaussian to represent an approximate belief state, as we discuss in Section 8.3, Section 8.4, Section 8.5 and Section 8.6. We discuss some non-Gaussian approximations in Section 8.7.

For most of this chapter, we assume the SSM can be written as a nonlinear model subject to additive Gaussian noise:

$$\begin{aligned} \mathbf{z}_t &= \mathbf{f}(\mathbf{z}_{t-1}, \mathbf{u}_t) + \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{h}(\mathbf{z}_t, \mathbf{u}_t) + \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \end{aligned} \quad (8.15)$$

where \mathbf{f} is the transition or dynamics function, and \mathbf{h} is the observation function. In some cases, we will further assume that these functions are linear.

8.2 Inference for linear-Gaussian SSMs

In this section, we discuss inference in SSMs where all the distributions are linear Gaussian. This is called a **linear Gaussian state space model (LG-SSM)** or a **linear dynamical system (LDS)**. We discuss such models in detail in Section 29.6, but in brief they have the following form:

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{u}_t) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \mathbf{z}_{t-1} + \mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t, \mathbf{Q}_t) \quad (8.16)$$

$$p(\mathbf{y}_t | \mathbf{z}_t, \mathbf{u}_t) = \mathcal{N}(\mathbf{y}_t | \mathbf{H}_t \mathbf{z}_t + \mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t, \mathbf{R}_t) \quad (8.17)$$

where $\mathbf{z}_t \in \mathbb{R}^{N_z}$ is the hidden state, $\mathbf{y}_t \in \mathbb{R}^{N_y}$ is the observation, and $\mathbf{u}_t \in \mathbb{R}^{N_u}$ is the input. (We have allowed the parameters to be time-varying, for later extensions that we will consider.) We often assume the means of the process noise and observation noise (i.e., the bias or offset terms) are zero, so $\mathbf{b}_t = \mathbf{0}$ and $\mathbf{d}_t = \mathbf{0}$. In addition, we often have no inputs, so $\mathbf{B}_t = \mathbf{D}_t = \mathbf{0}$. In this case, the model

simplifies to the following:³

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \mathbf{z}_{t-1}, \mathbf{Q}_t) \quad (8.18)$$

$$p(\mathbf{y}_t | \mathbf{z}_t) = \mathcal{N}(\mathbf{y}_t | \mathbf{H}_t \mathbf{z}_t, \mathbf{R}_t) \quad (8.19)$$

See Figure 8.1 for the graphical model.⁴

Note that an LG-SSM is just a special case of a Gaussian Bayes net (Section 4.2.3), so the entire joint distribution $p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T} | \mathbf{u}_{1:T})$ is a large multivariate Gaussian with $N_y N_z T$ dimensions. However, it has a special structure that makes it computationally tractable to use, as we show below. In particular, we will discuss the **Kalman filter** and **Kalman smoother**, that can perform exact filtering and smoothing in $O(T N_z^3)$ time.

8.2.1 Examples

Before diving into the theory, we give some motivating examples.

8.2.1.1 Tracking and state estimation

A common application of LG-SSMs is for **tracking** objects, such as airplanes or animals, from noisy measurements, such as radar or cameras. For example, suppose we want to track an object moving in 2d. (We discuss this example in more detail in Section 29.7.1.) The hidden state \mathbf{z}_t encodes the location, (x_{t1}, x_{t2}) , and the velocity, $(\dot{x}_{t1}, \dot{x}_{t2})$, of the moving object. The observation \mathbf{y}_t is a noisy version of the location. (The velocity is not observed but can be inferred from the change in location.) We assume that we obtain measurements with a sampling period of Δ . The new location is the old location plus Δ times the velocity, plus noise added to all terms:

$$\mathbf{z}_t = \underbrace{\begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{F}} \mathbf{z}_{t-1} + \mathbf{q}_t \quad (8.20)$$

where $\mathbf{q}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t)$. The observation extracts the location and adds noise:

$$\mathbf{y}_t = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{\mathbf{H}} \mathbf{z}_t + \mathbf{r}_t \quad (8.21)$$

where $\mathbf{r}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t)$.

Our goal is to use this model to estimate the unknown location (and velocity) of the object given the noisy observations. In particular, in the filtering problem, we want to compute $p(\mathbf{z}_t | \mathbf{y}_{1:t})$ in

3. Our notation is similar to [SS23], except he writes $p(\mathbf{x}_k | \mathbf{x}_{k-1}) = \mathcal{N}(\mathbf{x}_k | \mathbf{A}_{k-1} \mathbf{x}_{k-1}, \mathbf{Q}_{k-1})$ instead of $p(\mathbf{z}_t | \mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \mathbf{z}_{t-1}, \mathbf{Q}_t)$, and $p(\mathbf{y}_k | \mathbf{x}_k) = \mathcal{N}(\mathbf{y}_k | \mathbf{H}_k \mathbf{x}_k, \mathbf{R}_k)$ instead of $p(\mathbf{y}_t | \mathbf{z}_t) = \mathcal{N}(\mathbf{y}_t | \mathbf{H}_t \mathbf{z}_t, \mathbf{R}_t)$.

4. Note that, for some problems, the evolution of certain components of the state vector is deterministic, in which case the corresponding noise terms must be zero. To avoid singular covariance matrices, we can replace the dynamics noise $\mathbf{w}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t)$ with $\mathbf{G}_t \tilde{\mathbf{w}}_t$, where $\tilde{\mathbf{w}}_t \sim \mathcal{N}(\mathbf{0}, \tilde{\mathbf{Q}}_t)$, where $\tilde{\mathbf{Q}}_t$ is a smaller $N_q \times N_q$ psd matrix, and \mathbf{G}_t is a $N_y \times N_q$. In this case, the covariance of the noise becomes $\mathbf{Q}_t = \mathbf{G}_t \tilde{\mathbf{Q}}_t \mathbf{G}_t^\top$.

a recursive fashion. Figure 8.2(b) illustrates filtering for the linear Gaussian SSM applied to the noisy tracking data in Figure 8.2(a) (shown by the green dots). The filtered estimates are computed using the Kalman filter algorithm described in Section 8.2.2. The red line shows the posterior mean estimate of the location, and the black circles show the posterior covariance. We see that the estimated trajectory is less noisy than the raw data, since it incorporates prior knowledge about how the data was generated.

Another task of interest is the smoothing problem where we want to compute $p(\mathbf{z}_t|\mathbf{y}_{1:T})$ using an offline dataset. Figure 8.2(c) illustrates smoothing for the LG-SSM, implemented using the Kalman smoothing algorithm described in Section 8.2.3. We see that the resulting estimate is smoother, and that the posterior uncertainty is reduced (as visualized by the smaller confidence ellipses).

8.2.1.2 Online Bayesian linear regression (recursive least squares)

In Section 29.7.2 we discuss how to use the Kalman filter to recursively compute the exact posterior $p(\mathbf{w}|\mathcal{D}_{1:t})$ for a linear regression model in an online fashion. This is known as the recursive least squares algorithm. The basic idea is to treat the latent state to be the parameter values, $\mathbf{z}_t = \mathbf{w}$, and to define the non-stationary observation model as $p(\mathbf{y}_t|\mathbf{z}_t) = \mathcal{N}(y_t|\mathbf{x}_t^\top \mathbf{z}_t, \sigma^2)$, and the dynamics model as $p(\mathbf{z}_t|\mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t|\mathbf{z}_{t-1}, 0\mathbf{I})$.

8.2.1.3 Time series forecasting

In Section 29.12, we discuss how to use Kalman filtering to perform time series forecasting.

8.2.2 The Kalman filter

The **Kalman filter (KF)** is an algorithm for exact Bayesian filtering for linear Gaussian state space models. The resulting algorithm is the Gaussian analog of the HMM filter in Section 9.2.2. The belief state at time t is now given by $p(\mathbf{z}_t|\mathbf{y}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$, where we use the notation $\boldsymbol{\mu}_{t|t}$ and $\boldsymbol{\Sigma}_{t|t}$ to represent the posterior mean and covariance given $\mathbf{y}_{1:t}$.⁵ Since everything is Gaussian, we can perform the prediction and update steps in closed form, as we explain below (see Section 8.2.2.4 for the derivation).

8.2.2.1 Predict step

The one-step-ahead prediction for the hidden state, also called the **time update step**, is given by the following:

$$p(\mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \quad (8.22)$$

$$\boldsymbol{\mu}_{t|t-1} = \mathbf{F}_t \boldsymbol{\mu}_{t-1|t-1} + \mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t \quad (8.23)$$

$$\boldsymbol{\Sigma}_{t|t-1} = \mathbf{F}_t \boldsymbol{\Sigma}_{t-1|t-1} \mathbf{F}_t^\top + \mathbf{Q}_t \quad (8.24)$$

5. We represent the mean and covariance of the filtered belief state by $\boldsymbol{\mu}_{t|t}$ and $\boldsymbol{\Sigma}_{t|t}$, but some authors use the notation \mathbf{m}_t and \mathbf{P}_t instead. We represent the mean and covariance of the smoothed belief state by $\boldsymbol{\mu}_{t|T}$ and $\boldsymbol{\Sigma}_{t|T}$, but some authors use the notation \mathbf{m}_t^s and \mathbf{P}_t^s instead. Finally, we represent the mean and covariance of the one-step-ahead posterior predictive distribution, $p(\mathbf{z}_t|\mathbf{y}_{1:t-1})$, by $\boldsymbol{\mu}_{t|t-1}$ and $\boldsymbol{\Sigma}_{t|t-1}$, whereas some authors use \mathbf{m}_t^- and \mathbf{P}_t^- instead.

8.2.2.2 Update step

The update step (also called the **measurement update step**) can be computed using Bayes' rule, as follows:

$$p(\mathbf{z}_t | \mathbf{y}_{1:t}, \mathbf{u}_{1:t}) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \quad (8.25)$$

$$\hat{\mathbf{y}}_t = \mathbf{H}_t \boldsymbol{\mu}_{t|t-1} + \mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t \quad (8.26)$$

$$\mathbf{S}_t = \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t \quad (8.27)$$

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top \mathbf{S}_t^{-1} \quad (8.28)$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t (\mathbf{y}_t - \hat{\mathbf{y}}_t) \quad (8.29)$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \quad (8.30)$$

$$= \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^\top \quad (8.31)$$

where \mathbf{K}_t is the **Kalman gain matrix**. Note that $\hat{\mathbf{y}}_t$ is the expected observation, so $\mathbf{e}_t = \mathbf{y}_t - \hat{\mathbf{y}}_t$ is the **residual error**, also called the **innovation term**. The covariance of the observation is denoted by \mathbf{S}_t , and the cross covariance between the observation and state is denoted by $\mathbf{C}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top$. In practice, to compute the Kalman gain, we do not use $\mathbf{K}_t = \mathbf{C}_t \mathbf{S}_t^{-1}$, but instead we solve the linear system $\mathbf{K}_t \mathbf{S}_t = \mathbf{C}_t$.⁶

To understand the update step intuitively, note that the update for the latent mean, $\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t \mathbf{e}_t$, is the predicted new latent mean plus a correction factor, which is \mathbf{K}_t times the error signal \mathbf{e}_t . If $\mathbf{H}_t = \mathbf{I}$, then $\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{S}_t^{-1}$; in the scalar case, this becomes $k_t = \boldsymbol{\Sigma}_{t|t-1} / S_t$, which is the ratio between the variance of the prior (from the dynamics model) and the variance of the measurement, which we can interpret as an inverse signal to noise ratio. If we have a strong prior and/or very noisy sensors, $|\mathbf{K}_t|$ will be small, and we will place little weight on the correction term. Conversely, if we have a weak prior and/or high precision sensors, then $|\mathbf{K}_t|$ will be large, and we will place a lot of weight on the correction term. Similarly, the new covariance is the old covariance minus a positive definite matrix, which depends on how informative the measurement is.

Note that, by using the matrix inversion lemma, the Kalman gain matrix can also be written as

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top (\mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t)^{-1} = (\boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{H}_t^\top \mathbf{R}_t^{-1} \mathbf{H}_t)^{-1} \mathbf{H}_t^\top \mathbf{R}_t^{-1} \quad (8.32)$$

This is useful if \mathbf{R}_t^{-1} is precomputed (e.g., if it is constant over time) and $N_y \gg N_z$. In addition, in Equation (8.97), we give the information form of the filter, which shows that the posterior precision has the form $\boldsymbol{\Sigma}_t^{-1} = \boldsymbol{\Sigma}_{t|t-1}^{-1} + \mathbf{H}_t^\top \mathbf{R}_t^{-1} \mathbf{H}_t$, so we can also write the gain matrix as $\mathbf{K}_t = \boldsymbol{\Sigma}_t \mathbf{H}_t^\top \mathbf{R}_t^{-1}$.

8.2.2.3 Posterior predictive

The one-step-ahead posterior predictive density for the observations can be computed as follows. (We ignore inputs and bias terms, for notational brevity.) First we compute the one-step-ahead predictive density for latent states:

$$p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) = \int p(\mathbf{z}_t | \mathbf{z}_{t-1}) p(\mathbf{z}_{t-1} | \mathbf{y}_{1:t-1}) d\mathbf{z}_{t-1} \quad (8.33)$$

$$= \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \boldsymbol{\mu}_{t-1|t-1}, \mathbf{F}_t \boldsymbol{\Sigma}_{t-1|t-1} \mathbf{F}_t^\top + \mathbf{Q}_t) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \quad (8.34)$$

6. Equivalently we have $\mathbf{S}_t^\top \mathbf{K}_t^\top = \mathbf{C}_t^\top$, so we can compute \mathbf{K}_t in JAX using $\mathbf{K} = \text{jnp.linalg.lstsq}(\mathbf{S}.\mathbf{T}, \mathbf{C}.\mathbf{T})[0].\mathbf{T}$.

Then we convert this to a prediction about observations by marginalizing out \mathbf{z}_t :

$$p(\mathbf{y}_t | \mathbf{y}_{1:t-1}) = \int p(\mathbf{y}_t, \mathbf{z}_t | \mathbf{y}_{1:t-1}) d\mathbf{z}_t = \int p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) d\mathbf{z}_t = \mathcal{N}(\mathbf{y}_t | \hat{\mathbf{y}}_t, \mathbf{S}_t) \quad (8.35)$$

This can also be used to compute the log-likelihood of the observations: The normalization constant of the new posterior can be computed as follows:

$$\log p(\mathbf{y}_{1:T}) = \sum_{t=1}^T \log p(\mathbf{y}_t | \mathbf{y}_{1:t-1}) = \sum_{t=1}^T \log Z_t \quad (8.36)$$

where we define $p(\mathbf{y}_1 | \mathbf{y}_0) = p(\mathbf{y}_1)$. This is just a sum of the log probabilities of the one-step-ahead measurement predictions, and is a measure of how “surprised” the model is at each step.

We can generalize the prediction step to predict observations K steps into the future by first forecasting K steps in latent space, and then “grounding” the final state into predicted observations. (This is in contrast to an RNN (Section 16.3.4), which requires generating observations at each step, in order to update future hidden states.)

8.2.2.4 Derivation

In this section we derive the Kalman filter equations, following [SS23, Sec 6.3]. The results are a straightforward application of the rules for manipulating linear Gaussian systems, discussed in Section 2.3.2.

First we derive the prediction step. From Equation (2.120), the joint predictive distribution for states is given by

$$p(\mathbf{z}_{t-1}, \mathbf{z}_t | \mathbf{y}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{t-1}) p(\mathbf{z}_{t-1} | \mathbf{y}_{1:t-1}) \quad (8.37)$$

$$= \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \mathbf{z}_{t-1}, \mathbf{Q}_t) \mathcal{N}(\mathbf{z}_{t-1} | \boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}) \quad (8.38)$$

$$= \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_{t-1} \\ \mathbf{z}_t \end{pmatrix} | \boldsymbol{\mu}', \boldsymbol{\Sigma}'\right) \quad (8.39)$$

where

$$\boldsymbol{\mu}' = \begin{pmatrix} \boldsymbol{\mu}_{t-1|t-1} \\ \mathbf{F}_t \boldsymbol{\mu}_{t-1|t-1} \end{pmatrix}, \quad \boldsymbol{\Sigma}' = \begin{pmatrix} \boldsymbol{\Sigma}_{t-1|t-1} & \boldsymbol{\Sigma}_{t-1|t-1} \mathbf{F}_t^\top \\ \mathbf{F}_t \boldsymbol{\Sigma}_{t-1|t-1} & \mathbf{F}_t \boldsymbol{\Sigma}_{t-1|t-1} \mathbf{F}_t^\top + \mathbf{Q}_t \end{pmatrix} \quad (8.40)$$

Hence the marginal predictive distribution for states is given by

$$p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_t \boldsymbol{\mu}_{t-1|t-1}, \mathbf{F}_t \boldsymbol{\Sigma}_{t-1|t-1} \mathbf{F}_t^\top + \mathbf{Q}_t) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \quad (8.41)$$

Now we derive the measurement update step. The joint distribution for state and observation is given by

$$p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{y}_{1:t-1}) = p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) \quad (8.42)$$

$$= \mathcal{N}(\mathbf{y}_t | \mathbf{H}_t \mathbf{z}_t, \mathbf{R}_t) \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \quad (8.43)$$

$$= \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_t \\ \mathbf{y}_t \end{pmatrix} | \boldsymbol{\mu}'', \boldsymbol{\Sigma}''\right) \quad (8.44)$$

where

$$\boldsymbol{\mu}'' = \begin{pmatrix} \boldsymbol{\mu}_{t|t-1} \\ \mathbf{H}_t \boldsymbol{\mu}_{t|t-1} \end{pmatrix}, \quad \boldsymbol{\Sigma}'' = \begin{pmatrix} \boldsymbol{\Sigma}_{t|t-1} & \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top \\ \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} & \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1}^{-1} \mathbf{H}_t^\top + \mathbf{R}_t \end{pmatrix} \quad (8.45)$$

Finally, we convert this joint into a conditional using Equation (2.78) as follows:

$$p(\mathbf{z}_t | \mathbf{y}_t, \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \quad (8.46)$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top (\mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t)^{-1} [\mathbf{y}_t - \mathbf{H}_t \boldsymbol{\mu}_{t|t-1}] \quad (8.47)$$

$$= \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t [\mathbf{y}_t - \mathbf{H}_t \boldsymbol{\mu}_{t|t-1}] \quad (8.48)$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top (\mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t)^{-1} \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \quad (8.49)$$

$$= \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \quad (8.50)$$

where

$$\mathbf{S}_t = \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t \quad (8.51)$$

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top \mathbf{S}_t^{-1} \quad (8.52)$$

8.2.2.5 Abstract formulation

We can represent the Kalman filter equations much more compactly by defining various functions that create and manipulate jointly Gaussian systems, as in Section 2.3.2. In particular, suppose we have the following linear Gaussian system:

$$p(\mathbf{z}) = \mathcal{N}(\check{\boldsymbol{\mu}}, \check{\boldsymbol{\Sigma}}) \quad (8.53)$$

$$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{A}\mathbf{z} + \mathbf{b}, \boldsymbol{\Omega}) \quad (8.54)$$

Then the joint is given by

$$p(\mathbf{z}, \mathbf{y}) = \mathcal{N} \left(\begin{pmatrix} \check{\boldsymbol{\mu}} \\ \boldsymbol{\mu} \end{pmatrix}, \begin{pmatrix} \check{\boldsymbol{\Sigma}} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{S} \end{pmatrix} \right) = \mathcal{N} \left(\begin{pmatrix} \check{\boldsymbol{\mu}} \\ \mathbf{A}\check{\boldsymbol{\mu}} + \mathbf{b} \end{pmatrix}, \begin{pmatrix} \check{\boldsymbol{\Sigma}} & \check{\boldsymbol{\Sigma}} \mathbf{A}^\top \\ \mathbf{A} \check{\boldsymbol{\Sigma}} & \mathbf{A} \check{\boldsymbol{\Sigma}} \mathbf{A}^\top + \boldsymbol{\Omega} \end{pmatrix} \right) \quad (8.55)$$

and the posterior is given by

$$p(\mathbf{z}|\mathbf{y}) = \mathcal{N}(\mathbf{z} | \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}) = \mathcal{N} \left(\mathbf{z} | \check{\boldsymbol{\mu}} + \mathbf{K}(\mathbf{y} - \check{\boldsymbol{\mu}}), \check{\boldsymbol{\Sigma}} - \mathbf{K} \mathbf{S} \mathbf{K}^\top \right) \quad (8.56)$$

where $\mathbf{K} = \mathbf{C} \mathbf{S}^{-1}$. See Algorithm 8.1 for the pseudocode.

We can now apply these functions to derive Kalman filtering as follows. In the prediction step, we compute

$$p(\mathbf{z}_{t-1}, \mathbf{z}_t | \mathbf{y}_{1:t-1}) = \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu}_{t-1|t-1} \\ \boldsymbol{\mu}_{t|t-1} \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma}_{t-1|t-1} & \boldsymbol{\Sigma}_{t-1,t|t-1} \\ \boldsymbol{\Sigma}_{t,t-1|t-1} & \boldsymbol{\Sigma}_{t|t-1} \end{pmatrix} \right) \quad (8.57)$$

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \boldsymbol{\Sigma}_{t-1,t|t}) = \text{GaussMoments}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{F}_t, \mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t, \mathbf{Q}_t) \quad (8.58)$$

Algorithm 8.1: Functions for a linear Gaussian system.

```

1 def GaussMoments( $\bar{\mu}$ ,  $\bar{\Sigma}$ ,  $\mathbf{A}$ ,  $\mathbf{b}$ ,  $\Omega$ ) :
2    $\bar{\mu} = \mathbf{A} \bar{\mu} + \mathbf{b}$ 
3    $\mathbf{S} = \Omega + \mathbf{A} \bar{\Sigma} \mathbf{A}^\top$ 
4    $\mathbf{C} = \bar{\Sigma} \mathbf{A}^\top$ 
5   Return ( $\bar{\mu}$ ,  $\mathbf{S}$ ,  $\mathbf{C}$ )
6 def GaussCondition( $\bar{\mu}$ ,  $\bar{\Sigma}$ ,  $\bar{\mu}$ ,  $\mathbf{S}$ ,  $\mathbf{C}$ ,  $\mathbf{y}$ ) :
7    $\mathbf{K} = \mathbf{C} \mathbf{S}^{-1}$ 
8    $\hat{\mu} = \bar{\mu} + \mathbf{K}(\mathbf{y} - \bar{\mu})$ 
9    $\hat{\Sigma} = \bar{\Sigma} - \mathbf{K} \mathbf{S} \mathbf{K}^\top$ 
10   $\ell = \log \mathcal{N}(\mathbf{y} | \bar{\mu}, \mathbf{S})$ 
11  Return ( $\hat{\mu}$ ,  $\hat{\Sigma}$ ,  $\ell$ )

```

from which we get the marginal distribution

$$p(\mathbf{z}_t | \mathbf{y}_{1:t-1}) = \mathcal{N}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}) \quad (8.59)$$

In the update step, we compute the joint distribution

$$p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{y}_{1:t-1}) = \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_{t|t-1} \\ \hat{\mu}_t \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma}_{t|t-1} & \mathbf{C}_t \\ \mathbf{C}_t^\top & \mathbf{S}_t \end{pmatrix}\right) \quad (8.60)$$

$$(\hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t) = \text{GaussMoments}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{H}_t, \mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t, \mathbf{R}_t) \quad (8.61)$$

We then condition this on the observations to get the posterior distribution

$$p(\mathbf{z}_t | \mathbf{y}_t, \mathbf{y}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{y}_{1:t}) = \mathcal{N}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \quad (8.62)$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \text{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t, \mathbf{y}_t) \quad (8.63)$$

The overall KF algorithm is shown in Algorithm 8.2.

Algorithm 8.2: Kalman filter.

```

1 def KF( $\mathbf{F}_{1:T}$ ,  $\mathbf{B}_{1:T}$ ,  $\mathbf{b}_{1:T}$ ,  $\mathbf{Q}_{1:T}$ ,  $\mathbf{H}_{1:T}$ ,  $\mathbf{D}_{1:T}$ ,  $\mathbf{d}_{1:T}$ ,  $\mathbf{R}_{1:T}$ ,  $\mathbf{u}_{1:T}$ ,  $\mathbf{y}_{1:T}$ ,  $\boldsymbol{\mu}_{0|0}$ ,  $\boldsymbol{\Sigma}_{0|0}$ ) :
2   foreach  $t = 1 : T$  do
3     // Predict:
4      $(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \text{GaussMoments}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{F}_t, \mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t, \mathbf{Q}_t)$ 
5     // Update:
6      $(\bar{\mu}, \mathbf{S}, \mathbf{C}) = \text{GaussMoments}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{H}_t, \mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t, \mathbf{R}_t)$ 
7      $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \text{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \bar{\mu}, \mathbf{S}, \mathbf{C}, \mathbf{y})$ 
8   Return  $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})_{t=1}^T, \sum_{t=1}^T \ell_t$ 

```

8.2.2.6 Numerical issues

In practice, the Kalman filter can encounter numerical issues. One solution is to use the **information filter**, which recursively updates the natural parameters of the Gaussian, $\boldsymbol{\Lambda}_{t|t} = \boldsymbol{\Sigma}_{t|t}^{-1}$ and $\boldsymbol{\eta}_{t|t} = \boldsymbol{\Lambda}_t \boldsymbol{\mu}_{t|t}$, instead of the mean and covariance (see Section 8.2.4). Another solution is the **square root filter**, which works with the Cholesky or QR decomposition of $\boldsymbol{\Sigma}_{t|t}$, which is much more numerically stable than directly updating $\boldsymbol{\Sigma}_{t|t}$. These techniques can be combined to create the **square root information filter** (SRIF) [May79]. (According to [Bie06], the SRIF was developed in 1969 for use in JPL's Mariner 10 mission to Venus.) In [Tol22] they present an approach which uses QR decompositions instead of matrix inversions, which can also be more stable.

8.2.2.7 Continuous-time version

The Kalman filter can be extended to work with continuous time dynamical systems; the resulting method is called the **Kalman Bucy filter**. See [SS19, p208] for details. q

8.2.3 The Kalman (RTS) smoother

In Section 8.2.2, we described the Kalman filter, which sequentially computes $p(\mathbf{z}_t | \mathbf{y}_{1:t})$ for each t . This is useful for online inference problems, such as tracking. However, in an offline setting, we can wait until all the data has arrived, and then compute $p(\mathbf{z}_t | \mathbf{y}_{1:T})$. By conditioning on past and future data, our uncertainty will be significantly reduced. This is illustrated in Figure 8.2(c), where we see that the posterior covariance ellipsoids are smaller for the smoothed trajectory than for the filtered trajectory.

We now explain how to compute the smoothed estimates, using an algorithm called the **RTS smoother** or **RTSS**, named after its inventors, Rauch, Tung, and Striebel [RTS65]. It is also known as the **Kalman smoothing** algorithm. The algorithm is the linear-Gaussian analog to the forwards-filtering backwards-smoothing algorithm for HMMs in Section 9.2.4.

8.2.3.1 Algorithm

In this section, we state the Kalman smoother algorithm. We give the derivation in Section 8.2.3.2.

The key update equations are as follows: From this, we can extract the smoothed marginal

$$p(\mathbf{z}_t | \mathbf{y}_{1:T}) = \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) \quad (8.64)$$

$$\boldsymbol{\mu}_{t+1|t} = \mathbf{F}_t \boldsymbol{\mu}_{t|t} \quad (8.65)$$

$$\boldsymbol{\Sigma}_{t+1|t} = \mathbf{F}_t \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top + \mathbf{Q}_{t+1} \quad (8.66)$$

$$\mathbf{J}_t = \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top \boldsymbol{\Sigma}_{t+1|t}^{-1} \quad (8.67)$$

$$\boldsymbol{\mu}_{t|T} = \boldsymbol{\mu}_{t|t} + \mathbf{J}_t (\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t}) \quad (8.68)$$

$$\boldsymbol{\Sigma}_{t|T} = \boldsymbol{\Sigma}_{t|t} + \mathbf{J}_t (\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{t+1|t}) \mathbf{J}_t^\top \quad (8.69)$$

8.2.3.2 Derivation

In this section, we derive the RTS smoother, following [SS23, Sec 12.2]. As in the derivation of the Kalman filter in Section 8.2.2.4, we make heavy use of the rules for manipulating linear Gaussian

systems, discussed in Section 2.3.2.

The joint filtered distribution for two consecutive time slices is

$$p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}) = p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t}) = \mathcal{N}(\mathbf{z}_{t+1} | \mathbf{F}_t \mathbf{z}_t, \mathbf{Q}_{t+1}) \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}) \quad (8.70)$$

$$= \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_t \\ \mathbf{z}_{t+1} \end{pmatrix} | \mathbf{m}_1, \mathbf{V}_1\right) \quad (8.71)$$

where

$$\mathbf{m}_1 = \begin{pmatrix} \boldsymbol{\mu}_{t|t} \\ \mathbf{F}_t \boldsymbol{\mu}_{t|t} \end{pmatrix}, \quad \mathbf{V}_1 = \begin{pmatrix} \boldsymbol{\Sigma}_{t|t} & \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top \\ \mathbf{F}_t \boldsymbol{\Sigma}_{t|t} & \mathbf{F}_t \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top + \mathbf{Q}_{t+1} \end{pmatrix} \quad (8.72)$$

By the Markov property for the hidden states we have

$$p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:t}, \mathbf{y}_{t+1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:t}) \quad (8.73)$$

and hence by conditioning the joint distribution $p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t})$ on the future state we get

$$p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) = \mathcal{N}(\mathbf{z}_t | \mathbf{m}_2, \mathbf{V}_2) \quad (8.74)$$

$$\boldsymbol{\mu}_{t+1|t} = \mathbf{F}_t \boldsymbol{\mu}_{t|t} \quad (8.75)$$

$$\boldsymbol{\Sigma}_{t+1|t} = \mathbf{F}_t \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top + \mathbf{Q}_{t+1} \quad (8.76)$$

$$\mathbf{J}_t = \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top \boldsymbol{\Sigma}_{t+1|t}^{-1} \quad (8.77)$$

$$\mathbf{m}_2 = \boldsymbol{\mu}_{t|t} + \mathbf{J}_t (\mathbf{z}_{t+1} - \boldsymbol{\mu}_{t+1|t}) \quad (8.78)$$

$$\mathbf{V}_2 = \boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t \boldsymbol{\Sigma}_{t+1|t} \mathbf{J}_t^\top \quad (8.79)$$

where \mathbf{J}_t is the backwards Kalman gain matrix.

$$\mathbf{J}_t = \boldsymbol{\Sigma}_{t,t+1|t} \boldsymbol{\Sigma}_{t+1|t}^{-1} \quad (8.80)$$

where $\boldsymbol{\Sigma}_{t,t+1|t} = \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top$ is the cross covariance term in the upper right block of \mathbf{V}_1 .

The joint distribution of two consecutive time slices given all the data is

$$p(\mathbf{z}_{t+1}, \mathbf{z}_t | \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T}) \quad (8.81)$$

$$= \mathcal{N}(\mathbf{z}_t | \mathbf{m}_2(\mathbf{z}_{t+1}), \mathbf{V}_2) \mathcal{N}(\mathbf{z}_{t+1} | \boldsymbol{\mu}_{t+1|T}, \boldsymbol{\Sigma}_{t+1|T}) \quad (8.82)$$

$$= \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_{t+1} \\ \mathbf{z}_t \end{pmatrix} | \mathbf{m}_3, \mathbf{V}_3\right) \quad (8.83)$$

where

$$\mathbf{m}_3 = \begin{pmatrix} \boldsymbol{\mu}_{t+1|T} \\ \boldsymbol{\mu}_{t|t} + \mathbf{J}_t (\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t}) \end{pmatrix}, \quad \mathbf{V}_3 = \begin{pmatrix} \boldsymbol{\Sigma}_{t+1|T} & \boldsymbol{\Sigma}_{t+1|T} \mathbf{J}_t^\top \\ \mathbf{J}_t \boldsymbol{\Sigma}_{t+1|T} & \mathbf{J}_t \boldsymbol{\Sigma}_{t+1|T} \mathbf{J}_t^\top + \mathbf{V}_2 \end{pmatrix} \quad (8.84)$$

From this, we can extract $p(\mathbf{z}_t | \mathbf{y}_{1:T})$, with the mean and covariance given by Equation (8.68) and Equation (8.69).

8.2.3.3 Two-filter smoothing

Note that the backwards pass of the Kalman smoother does not need access to the observations, $\mathbf{y}_{1:T}$, but does need access to the filtered belief states from the forwards pass, $p(\mathbf{z}_t|\mathbf{y}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$. There is an alternative version of the algorithm, known as **two-filter smoothing** [FP69; Kit04], in which we compute the forwards pass as usual, and then separately compute backwards messages $p(\mathbf{y}_{t+1:T}|\mathbf{z}_t) \propto \mathcal{N}(\mathbf{y}_{t+1:T}|\boldsymbol{\mu}_{t|t}^b, \boldsymbol{\Sigma}_{t|t}^b)$, similar to the backwards filtering algorithm in HMMs (Section 9.2.3).

However, these backwards messages are conditional likelihoods, not posteriors, which can cause numerical problems. For example, consider $t = T$; in this case, we need to set the initial covariance matrix to be $\boldsymbol{\Sigma}_T^b = \infty \mathbf{I}$, so that the backwards message has no effect on the filtered posterior (since there is no evidence beyond step T). This problem can be resolved by working in information form. An alternative approach is to generalize the two-filter smoothing equations to ensure the likelihoods are normalizable by multiplying them by artificial distributions [BDM10].

In general, the RTS smoother is preferred to the two-filter smoother, since it is more numerically stable, and it is easier to generalize it to the nonlinear case.

8.2.3.4 Time and space complexity

In general, the Kalman smoothing algorithm takes $O(N_y^3 + N_z^2 + N_y N_z)$ per step, where there are T steps. This can be slow when applied to long sequences. In [SGF21], they describe how to reduce this to $O(\log T)$ steps using a **parallel prefix scan** operator that can be run efficiently on GPUs. In addition, we can reduce the space from $O(T)$, to $O(\log T)$ using the same algorithm as in Section 9.2.5.

8.2.3.5 Forwards filtering backwards sampling

To draw posterior samples from the LG-SSM, we can leverage the following result:

$$p(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{y}_{1:T}) = \mathcal{N}(\mathbf{z}_t|\tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t) \quad (8.85)$$

$$\tilde{\boldsymbol{\mu}}_t = \boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\mathbf{z}_{t+1} - \mathbf{F}_t \boldsymbol{\mu}_{t|t}) \quad (8.86)$$

$$\tilde{\boldsymbol{\Sigma}}_t = \boldsymbol{\Sigma}_{t|t} - \mathbf{J}_t \boldsymbol{\Sigma}_{t+1|t} \mathbf{J}_t^\top = \boldsymbol{\Sigma}_{t|t} - \boldsymbol{\Sigma}_{t|t} \mathbf{F}_t^\top \boldsymbol{\Sigma}_{t+1|t}^{-1} \boldsymbol{\Sigma}_{t+1|t} \mathbf{J}_t^\top \quad (8.87)$$

$$= \boldsymbol{\Sigma}_{t|t} (\mathbf{I} - \mathbf{F}_t^\top \mathbf{J}_t^\top) \quad (8.88)$$

where \mathbf{J}_t is the backwards Kalman gain defined in Equation (8.67).

8.2.4 Information form filtering and smoothing

This section is written by Giles Harper-Donnelly.

In this section, we derive the Kalman filter and smoother algorithms in information form. We will see that this is the “dual” of Kalman filtering/smoothing in moment form. In particular, while computing marginals in moment form is easy, computing conditionals is hard (requires a matrix inverse). Conversely, for information form, computing marginals is hard, but computing conditionals is easy.

8.2.4.1 Filtering: algorithm

The predict step has a similar structure to the update step in moment form. We start with the prior $p(\mathbf{z}_{t-1}|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t-1}) = \mathcal{N}_c(\mathbf{z}_{t-1}|\boldsymbol{\eta}_{t-1|t-1}, \boldsymbol{\Lambda}_{t-1|t-1})$ and then compute

$$p(\mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = \mathcal{N}_c(\mathbf{z}_t|\boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}) \quad (8.89)$$

$$\mathbf{M}_t = \boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\top \mathbf{Q}_t^{-1} \mathbf{F}_t \quad (8.90)$$

$$\mathbf{J}_t = \mathbf{Q}_t^{-1} \mathbf{F}_t \mathbf{M}_t^{-1} \quad (8.91)$$

$$\boldsymbol{\Lambda}_{t|t-1} = \mathbf{Q}_t^{-1} - \mathbf{Q}_t^{-1} \mathbf{F}_t (\boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\top \mathbf{Q}_t^{-1} \mathbf{F}_t)^{-1} \mathbf{F}_t^\top \mathbf{Q}_t^{-1} \quad (8.92)$$

$$= \mathbf{Q}_t^{-1} - \mathbf{J}_t \mathbf{F}_t^\top \mathbf{Q}_t^{-1} \quad (8.93)$$

$$= \mathbf{Q}_t^{-1} - \mathbf{J}_t \mathbf{M}_t \mathbf{J}_t^\top \quad (8.94)$$

$$\boldsymbol{\eta}_{t|t-1} = \mathbf{J}_t \boldsymbol{\eta}_{t-1|t-1} + \boldsymbol{\Lambda}_{t|t-1} (\mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t), \quad (8.95)$$

where \mathbf{J}_t is analogous to the Kalman gain matrix in moment form Equation (8.28). From the matrix inversion lemma, Equation (2.93), we see that Equation (8.92) is the inverse of the predicted covariance $\boldsymbol{\Sigma}_{t|t-1}$ given in Equation (8.24).

The update step in information form is as follows:

$$p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}) = \mathcal{N}_c(\mathbf{z}_t|\boldsymbol{\eta}_{t|t}, \boldsymbol{\Lambda}_{t|t}) \quad (8.96)$$

$$\boldsymbol{\Lambda}_{t|t} = \boldsymbol{\Lambda}_{t|t-1} + \mathbf{H}_t^\top \mathbf{R}_t^{-1} \mathbf{H}_t \quad (8.97)$$

$$\boldsymbol{\eta}_{t|t} = \boldsymbol{\eta}_{t|t-1} + \mathbf{H}_t^\top \mathbf{R}_t^{-1} (\mathbf{y}_t - \mathbf{D}_t \mathbf{u}_t - \mathbf{d}_t). \quad (8.98)$$

8.2.4.2 Filtering: derivation

For the predict step, we first derive the joint distribution over hidden states at $t, t-1$:

$$p(\mathbf{z}_{t-1}, \mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = p(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{u}_t) p(\mathbf{z}_{t-1}|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t-1}) \quad (8.99)$$

$$= \mathcal{N}_c(\mathbf{z}_t, |\mathbf{Q}_t^{-1} (\mathbf{F}_t \mathbf{z}_{t-1} + \mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t), \mathbf{Q}_t^{-1}) \quad (8.100)$$

$$\times \mathcal{N}_c(\mathbf{z}_{t-1}, |\boldsymbol{\eta}_{t-1|t-1}, \boldsymbol{\Lambda}_{t-1|t-1}) \quad (8.101)$$

$$= \mathcal{N}_c(\mathbf{z}_{t-1}, \mathbf{z}_t|\boldsymbol{\eta}_{t-1,t|t}, \boldsymbol{\Lambda}_{t-1,t|t}) \quad (8.102)$$

where

$$\boldsymbol{\eta}_{t-1,t|t-1} = \begin{pmatrix} \boldsymbol{\eta}_{t-1|t-1} - \mathbf{F}_t^\top \mathbf{Q}_t^{-1} (\mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t) \\ \mathbf{Q}_t^{-1} (\mathbf{B}_t \mathbf{u}_t + \mathbf{b}_t) \end{pmatrix} \quad (8.103)$$

$$\boldsymbol{\Lambda}_{t-1,t|t-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{t-1|t-1} + \mathbf{F}_t^\top \mathbf{Q}_t^{-1} \mathbf{F}_t & -\mathbf{F}_t^\top \mathbf{Q}_t^{-1} \\ -\mathbf{Q}_t^{-1} \mathbf{F}_t & \mathbf{Q}_t^{-1} \end{pmatrix} \quad (8.104)$$

The information form predicted parameters $\boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}$ can then be derived using the marginalisation formulae in Section 2.3.1.4.

For the update step, we start with the joint distribution over the hidden state and the observation

at t :

$$p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = p(\mathbf{y}_t | \mathbf{z}_t, \mathbf{u}_t) p(\mathbf{z}_t | \mathbf{y}_{1:t-1}, \mathbf{u}_{1:t-1}) \quad (8.105)$$

$$= \mathcal{N}_c(\mathbf{y}_t, |\mathbf{R}_t^{-1}(\mathbf{H}_t \mathbf{z}_t + \mathbf{D} \mathbf{u}_t + \mathbf{d}_t), \mathbf{R}_t^{-1}) \mathcal{N}_c(\mathbf{z}_t | \boldsymbol{\eta}_{t|t-1}, \boldsymbol{\Lambda}_{t|t-1}) \quad (8.106)$$

$$= \mathcal{N}_c(\mathbf{z}_t, \mathbf{y} | \boldsymbol{\eta}_{z,y|t}, \boldsymbol{\Lambda}_{z,y|t}) \quad (8.107)$$

where

$$\boldsymbol{\eta}_{z,y|t} = \begin{pmatrix} \boldsymbol{\eta}_{t|t-1} - \mathbf{H}_t^T \mathbf{R}_t^{-1} (\mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t) \\ \mathbf{R}_t^{-1} (\mathbf{D}_t \mathbf{u}_t + \mathbf{d}_t) \end{pmatrix} \quad (8.108)$$

$$\boldsymbol{\Lambda}_{z,y|t} = \begin{pmatrix} \boldsymbol{\Lambda}_{t|t-1} + \mathbf{H}_t^T \mathbf{R}_t^{-1} \mathbf{H}_t & -\mathbf{H}_t^T \mathbf{R}_t^{-1} \\ -\mathbf{R}_t^{-1} \mathbf{H}_t & \mathbf{R}_t^{-1} \end{pmatrix} \quad (8.109)$$

The information form filtered parameters $\boldsymbol{\eta}_{t|t}$, $\boldsymbol{\Lambda}_{t|t}$ are then derived using the conditional formulae in 2.3.1.4.

8.2.4.3 Smoothing: algorithm

The smoothing equations are as follows:

$$p(\mathbf{z}_t | \mathbf{y}_{1:T}) = \mathcal{N}_c(\mathbf{z}_t | \boldsymbol{\eta}_{t|T}, \boldsymbol{\Lambda}_{t|T}) \quad (8.110)$$

$$\mathbf{U}_t = \mathbf{Q}_t^{-1} + \boldsymbol{\Lambda}_{t+1|T} - \boldsymbol{\Lambda}_{t+1|t} \quad (8.111)$$

$$\mathbf{L}_t = \mathbf{F}_t^T \mathbf{Q}_t^{-1} \mathbf{U}_t^{-1} \quad (8.112)$$

$$\boldsymbol{\Lambda}_{t|T} = \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_t^T \mathbf{Q}_t^{-1} \mathbf{F}_t - \mathbf{L}_t \mathbf{Q}_t^{-1} \mathbf{F} \quad (8.113)$$

$$= \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_t^T \mathbf{Q}_t^{-1} \mathbf{F}_t - \mathbf{L}_t \mathbf{U}_t \mathbf{L}_t^T \quad (8.114)$$

$$\boldsymbol{\eta}_{t|T} = \boldsymbol{\eta}_{t|t} + \mathbf{L}_t (\boldsymbol{\eta}_{t+1|T} - \boldsymbol{\eta}_{t+1|t}). \quad (8.115)$$

The parameters $\boldsymbol{\eta}_{t|t}$ and $\boldsymbol{\Lambda}_{t|t}$ are the filtered values from Equations (8.98) and (8.97) respectively. Similarly, $\boldsymbol{\eta}_{t+1|t}$ and $\boldsymbol{\Lambda}_{t+1|t}$ are the predicted parameters from Equations (8.95) and (8.92). The matrix \mathbf{L}_t is the information form analog to the backwards Kalman gain matrix in Equation (8.67).

8.2.4.4 Smoothing: derivation

From the generic forwards-filtering backwards-smoothing equation, Equation (8.14), we have

$$p(\mathbf{z}_t | \mathbf{y}_{1:T}) = p(\mathbf{z}_t | \mathbf{y}_{1:t}) \int \left[\frac{p(\mathbf{z}_{t+1} | \mathbf{z}_t) p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} \right] d\mathbf{z}_{t+1} \quad (8.116)$$

$$= \int p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}) \frac{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} | \mathbf{y}_{1:t})} d\mathbf{z}_{t+1} \quad (8.117)$$

$$= \int \mathcal{N}_c(\mathbf{z}_t, \mathbf{z}_{t+1} | \boldsymbol{\eta}_{t,t+1|t}, \boldsymbol{\Lambda}_{t,t+1|t}) \frac{\mathcal{N}_c(\mathbf{z}_{t+1} | \boldsymbol{\eta}_{t+1|T}, \boldsymbol{\Lambda}_{t+1|T})}{\mathcal{N}_c(\mathbf{z}_{t+1} | \boldsymbol{\eta}_{t+1|t}, \boldsymbol{\Lambda}_{t+1|t})} d\mathbf{z}_{t+1} \quad (8.118)$$

$$= \int \mathcal{N}_c(\mathbf{z}_t, \mathbf{z}_{t+1} | \boldsymbol{\eta}_{t,t+1|T}, \boldsymbol{\Lambda}_{t,t+1|T}) d\mathbf{z}_{t+1}. \quad (8.119)$$

The parameters of the joint filtering predictive distribution, $p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t})$, take precisely the same form as those in the filtering derivation described in Section 8.2.4.2:

$$\boldsymbol{\eta}_{t,t+1|t} = \begin{pmatrix} \boldsymbol{\eta}_{t|t} \\ \mathbf{0} \end{pmatrix}, \quad \boldsymbol{\Lambda}_{t,t+1|t} = \begin{pmatrix} \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_{t+1}^T \mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & -\mathbf{F}_{t+1}^T \mathbf{Q}_{t+1}^{-1} \\ -\mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & \mathbf{Q}_{t+1}^{-1} \end{pmatrix}, \quad (8.120)$$

We can now update this potential function by subtracting out the filtered information and adding in the smoothing information, using the rules for manipulating Gaussian potentials described in Section 2.3.3:

$$\boldsymbol{\eta}_{t,t+1|T} = \boldsymbol{\eta}_{t,t+1|t} + \begin{pmatrix} \mathbf{0} \\ \boldsymbol{\eta}_{t+1|T} \end{pmatrix} - \begin{pmatrix} \mathbf{0} \\ \boldsymbol{\eta}_{t+1|t} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\eta}_{t|t} \\ \boldsymbol{\eta}_{t+1|T} - \boldsymbol{\eta}_{t+1|t} \end{pmatrix}, \quad (8.121)$$

and

$$\boldsymbol{\Lambda}_{t,t+1|T} = \boldsymbol{\Lambda}_{t,t+1|t} + \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Lambda}_{t+1|T} \end{pmatrix} - \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Lambda}_{t+1|t} \end{pmatrix} \quad (8.122)$$

$$= \begin{pmatrix} \boldsymbol{\Lambda}_{t|t} + \mathbf{F}_{t+1}^T \mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & -\mathbf{F}_{t+1}^T \mathbf{Q}_{t+1}^{-1} \\ -\mathbf{Q}_{t+1}^{-1} \mathbf{F}_{t+1} & \mathbf{Q}_{t+1}^{-1} + \boldsymbol{\Lambda}_{t+1|T} - \boldsymbol{\Lambda}_{t+1|t} \end{pmatrix} \quad (8.123)$$

Applying the information from marginalization formula Equation (2.85) leads to Equation (8.115) and Equation (8.113).

8.3 Inference based on local linearization

In this section, we extend the Kalman filter and smoother to the case where the system dynamics and/or the observation model are nonlinear. (We continue to assume that the noise is additive Gaussian, as in Equation (8.15).) The basic idea is to linearize the dynamics and observation models about the previous state estimate using a first order Taylor series expansion, and then to apply the standard Kalman filter equations from Section 8.2.2. Intuitively we can think of this as approximating a stationary non-linear dynamical system with a non-stationary linear dynamical system. This approach is called the **extended Kalman filter** or **EKF**.

8.3.1 Taylor series expansion

Suppose $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\mathbf{y} = \mathbf{g}(\mathbf{x})$, where $\mathbf{g} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a differentiable and invertible function. The pdf for \mathbf{y} is given by

$$p(\mathbf{y}) = |\det \text{Jac}(\mathbf{g}^{-1})(\mathbf{y})| \mathcal{N}(\mathbf{g}^{-1}(\mathbf{y}) | \boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (8.124)$$

In general this is intractable to compute, so we seek an approximation.

Suppose $\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\delta}$, where $\boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. Then we can form a first order Taylor series expansion of the function \mathbf{g} as follows:

$$\mathbf{g}(\mathbf{x}) = \mathbf{g}(\boldsymbol{\mu} + \boldsymbol{\delta}) \approx \mathbf{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} \quad (8.125)$$

where $\mathbf{G}(\boldsymbol{\mu}) = \text{Jac}(\mathbf{g})(\boldsymbol{\mu})$ is the Jacobian of \mathbf{g} at $\boldsymbol{\mu}$:

$$[\mathbf{G}(\boldsymbol{\mu})]_{jj'} = \frac{\partial g_j(\mathbf{x})}{\partial x_{j'}}|_{\mathbf{x}=\boldsymbol{\mu}} \quad (8.126)$$

We now derive the induced Gaussian approximation to $\mathbf{y} = \mathbf{g}(\mathbf{x})$. The mean is given by

$$\mathbb{E}[\mathbf{y}] \approx \mathbb{E}[\mathbf{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta}] = \mathbf{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\mathbb{E}[\boldsymbol{\delta}] = \mathbf{g}(\boldsymbol{\mu}) \quad (8.127)$$

The covariance is given by

$$\text{Cov}[\mathbf{y}] = \mathbb{E}[(\mathbf{g}(\mathbf{x}) - \mathbb{E}[\mathbf{g}(\mathbf{x})])(\mathbf{g}(\mathbf{x}) - \mathbb{E}[\mathbf{g}(\mathbf{x})])^\top] \quad (8.128)$$

$$\approx \mathbb{E}[(\mathbf{g}(\mathbf{x}) - \mathbf{g}(\boldsymbol{\mu}))(\mathbf{g}(\mathbf{x}) - \mathbf{g}(\boldsymbol{\mu}))^\top] \quad (8.129)$$

$$\approx \mathbb{E}[(\mathbf{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} - \mathbf{g}(\boldsymbol{\mu}))(\mathbf{g}(\boldsymbol{\mu}) + \mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta} - \mathbf{g}(\boldsymbol{\mu}))^\top] \quad (8.130)$$

$$= \mathbb{E}[(\mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta})(\mathbf{G}(\boldsymbol{\mu})\boldsymbol{\delta})^\top] \quad (8.131)$$

$$= \mathbf{G}(\boldsymbol{\mu})\mathbb{E}[\boldsymbol{\delta}\boldsymbol{\delta}^\top]\mathbf{G}(\boldsymbol{\mu})^\top \quad (8.132)$$

$$= \mathbf{G}(\boldsymbol{\mu})\Sigma\mathbf{G}(\boldsymbol{\mu})^\top \quad (8.133)$$

Algorithm 8.3: Linearized approximation to a joint Gaussian distribution.

```

1 def LinearizedMoments( $\boldsymbol{\mu}, \Sigma, \mathbf{g}, \Omega$ ) :
2    $\hat{\mathbf{y}} = \mathbf{g}(\boldsymbol{\mu})$ 
3    $\mathbf{G} = \text{Jac}(\mathbf{g})(\boldsymbol{\mu})$ 
4    $\mathbf{S} = \mathbf{G}\Sigma\mathbf{G}^\top + \Omega$ 
5    $\mathbf{C} = \Sigma\mathbf{G}^\top$ 
6   Return  $(\hat{\mathbf{y}}, \mathbf{S}, \mathbf{C})$ 

```

When deriving the EKF, we need to compute the joint distribution $p(\mathbf{x}, \mathbf{y})$ where

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma), \mathbf{y} = \mathbf{g}(\mathbf{x}) + \mathbf{q}, \mathbf{q} \sim \mathcal{N}(\mathbf{0}, \Omega) \quad (8.134)$$

where \mathbf{q} is independent of \mathbf{x} . We can compute this by defining the augmented function $\tilde{\mathbf{g}}(\mathbf{x}) = [\mathbf{x}, \mathbf{g}(\mathbf{x})]$ and following the procedure above. The resulting linear approximation to the joint is

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu} \\ \hat{\mathbf{y}} \end{pmatrix}, \begin{pmatrix} \Sigma & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{S} \end{pmatrix} \right) = \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu} \\ \mathbf{g}(\boldsymbol{\mu}) \end{pmatrix}, \begin{pmatrix} \Sigma & \Sigma\mathbf{G}^\top \\ \mathbf{G}\Sigma & \mathbf{G}\Sigma\mathbf{G}^\top + \Omega \end{pmatrix} \right) \quad (8.135)$$

where the parameters are computed using Algorithm 8.3. We can then condition this joint Gaussian on the observed value \mathbf{y} to get the posterior.

It is also possible to derive an approximation for the case of non-additive Gaussian noise, where $\mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{q})$. See [SS23, Sec 7.1] for details.

8.3.2 The extended Kalman filter (EKF)

We now derive the extended Kalman filter for performing approximate inference in the model given by Equation (8.15). We first linearize the dynamics model around $\boldsymbol{\mu}_{t-1|t-1}$ to get an approximation to the one-step-ahead predictive distribution $p(\mathbf{z}_t|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t}) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t-1}, \Sigma_{t|t-1})$. We then linearize the observation model around $\boldsymbol{\mu}_{t|t-1}$, and then perform a Gaussian update. (In Section 8.3.2.2, we consider linearizing around a different point that gives better accuracy.)

We can write one step of the EKF algorithm using the notation from Section 8.2.2.5 as follows:

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \text{LinearizedMoments}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{f}(\cdot, \mathbf{u}_t), \mathbf{Q}_t) \quad (8.136)$$

$$(\hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t) = \text{LinearizedMoments}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{h}(\cdot, \mathbf{u}_t), \mathbf{R}_t) \quad (8.137)$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \text{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t, \mathbf{y}_t) \quad (8.138)$$

Spelling out the details more explicitly, we can write the predict step as follows:

$$\boldsymbol{\mu}_{t|t-1} = \mathbf{f}(\boldsymbol{\mu}_{t-1}, \mathbf{u}_t) \quad (8.139)$$

$$\boldsymbol{\Sigma}_{t|t-1} = \mathbf{F}_t \boldsymbol{\Sigma}_{t-1} \mathbf{F}_t^\top + \mathbf{Q}_t \quad (8.140)$$

where $\mathbf{F}_t \equiv \text{Jac}(f(\cdot, \mathbf{u}_t))(\boldsymbol{\mu}_{t|t-1})$ is the $N_z \times N_z$ Jacobian matrix of the dynamics model. The update step is as follows:

$$\hat{\mathbf{y}}_t = \mathbf{h}(\boldsymbol{\mu}_{t|t-1}, \mathbf{u}_t) \quad (8.141)$$

$$\mathbf{S}_t = \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top + \mathbf{R}_t \quad (8.142)$$

$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\top \mathbf{S}_t^{-1} \quad (8.143)$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t (\mathbf{y}_t - \hat{\mathbf{y}}_t) \quad (8.144)$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^\top \quad (8.145)$$

where $\mathbf{H}_t \equiv \text{Jac}(h(\cdot, \mathbf{u}_t))(\boldsymbol{\mu}_{t|t-1})$ is the $N_y \times N_z$ Jacobian matrix of the observation model and \mathbf{K}_t is the $N_z \times N_y$ Kalman gain matrix. See [Supplementary](#) Section 8.2.1 for the details of the derivation.

8.3.2.1 Accuracy

The EKF is widely used because it is simple and relatively efficient. However, there are two cases when the EKF works poorly [IX00; VDMW03]. The first is when the prior covariance is large. In this case, the prior distribution is broad, so we end up sending a lot of probability mass through different parts of the function that are far from $\boldsymbol{\mu}_{t-1|t-1}$, where the function has been linearized. The other setting where the EKF works poorly is when the function is highly nonlinear near the current mean (see Figure 8.5a).

A more accurate approach is to use a second-order Taylor series approximation, known as the **second order EKF**. The resulting updates can still be computed in closed form (see [SS23, Sec 7.3] for details). We can further improve performance by repeatedly re-linearizing the equations around $\boldsymbol{\mu}_t$ instead of $\boldsymbol{\mu}_{t|t-1}$; this is called the iterated EKF (see Section 8.3.2.2). In Section 8.4.2, we will discuss an algorithm called the unscented Kalman filter (UKF) which is even more accurate, and is derivative free (does not require computing Jacobians).

8.3.2.2 Iterated EKF

Another way to improve the accuracy of the EKF is by repeatedly re-linearizing the measurement model around the current posterior, $\boldsymbol{\mu}_{t|t}$, instead of $\boldsymbol{\mu}_{t|t-1}$; this is called the **iterated EKF** [BC93]. See Algorithm 8.4 for the pseudocode. (If we set the number of iterations to $J = 1$, we recover the standard EKF.)

Algorithm 8.4: Iterated extended Kalman filter.

```

1 def IEKF( $f, Q, h, R, y_{1:T}, \mu_{0|0}, \Sigma_{0|0}, J$ ) :
2   foreach  $t = 1 : T$  do
3     Predict step:
4        $(\mu_{t|t-1}, \Sigma_{t|t-1}, -) = \text{LinearizedMoments}(\mu_{t-1|t-1}, \Sigma_{t-1|t-1}, f(\cdot, u_t), Q_t)$ 
5     Update step:
6        $\mu_{t|t} = \mu_{t|t-1}, \Sigma_{t|t} = \Sigma_{t|t-1}$ 
7     foreach  $j = 1 : J$  do
8        $(\hat{y}_t, S_t, C_t) = \text{LinearizedMoments}(\mu_{t|t}, \Sigma_{t|t}, h(\cdot, u_t), R_t)$ 
9        $(\mu_{t|t}, \Sigma_{t|t}, \ell_t) = \text{GaussCondition}(\mu_{t|t-1}, \Sigma_{t|t-1}, \hat{y}_t, S_t, C_t, y_t)$ 
9   Return  $(\mu_{t|t}, \Sigma_{t|t})_{t=1}^T$ 

```

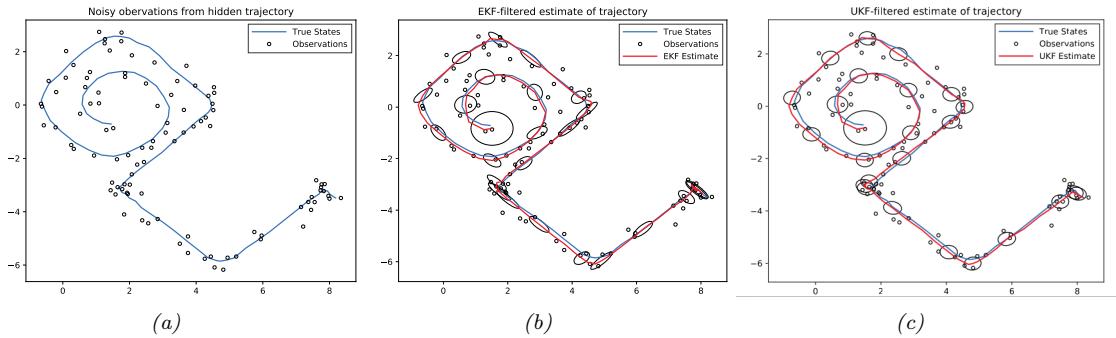


Figure 8.4: Illustration of filtering applied to a 2d nonlinear dynamical system. (a) True underlying state and observed data. (b) Extended Kalman filter estimate. Generated by [ekf_spiral.ipynb](#). (c) Unscented Kalman filter estimate. Generated by [ukf_spiral.ipynb](#).

The IEKF can be interpreted as a Gauss–Newton method for finding MAP estimate of the state at each step [BC93]. Specifically it minimizes the following objective:

$$\mathcal{L}(z_t) = \frac{1}{2}(\mathbf{y}_t - \mathbf{h}(z_t))^T \mathbf{R}_t^{-1} (\mathbf{y}_t - \mathbf{h}(z_t)) + \frac{1}{2}(z_t - \mu_{t|t-1})^T \Sigma_{t|t-1}^{-1} (z_t - \mu_{t|t-1}) \quad (8.146)$$

See [SS23, Sec 7.4] for details.

Unfortunately the Gauss–Newton method can sometimes diverge. Various robust extensions — including Levenberg–Marquardt, line search, and quasi–Newton methods — have been proposed in [SHA15; SS20a]. See [SS23, Sec 7.5] for details.

8.3.2.3 Example: Tracking a point spiraling in 2d

In Section 8.2.1.1, we considered an example of state estimation and tracking of an object moving in 2d under a linear dynamics model with a linear observation model. However, motion and observation models are often nonlinear. For example, consider an object that is moving along a curved trajectory,

such as this:

$$\mathbf{f}(\mathbf{z}) = (z_1 + \Delta \sin(z_2), z_2 + \Delta \cos(z_1)) \quad (8.147)$$

where Δ is the discrete step size (see [SS19, p221] for the continuous time version). For simplicity, we assume full visibility of the state vector (modulo observation noise), so $\mathbf{h}(\mathbf{z}) = \mathbf{z}$.

Despite the simplicity of this model, exact inference is intractable. However, we can easily apply the EKF. The results are shown in Figure 8.4b.

8.3.2.4 Example: Neural network training

In Section 17.5.2, we show how to use the EKF to perform online parameter inference for an MLP regression model.

8.3.3 The extended Kalman smoother (EKS)

We can extend the EKF to the offline smoothing case, resulting in the **extended Kalman smoother**, also called the **extended RTS smoother**. We just need to linearize the dynamics around the filtered mean when computing \mathbf{F}_t , and then we can apply the standard Kalman smoother update. See [SS23, Sec 13.1] for more details.

For improved accuracy, we can use the **iterated EKS**, which relinearizes the model at the previous MAP estimate. In [Bel94], they show that IEKS is equivalent to a Gauss-Newton method for computing the MAP estimate of the smoothing posterior. Unfortunately the IEKS can diverge in some cases. A robust IEKS method, that uses line search and Levenberg-Marquardt to update the parameters, is presented in [SS20a].

8.4 Inference based on the unscented transform

In this section, we replace the local linearization of the model with a different approximation. The key idea is this: instead of computing a linear approximation to the dynamics and measurement functions, and then passing a Gaussian distribution through the linearized functions, we instead approximate the joint distributions $p(\mathbf{z}_{t-1}, \mathbf{z}_t | \mathbf{y}_{1:t-1})$ and $p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{y}_{1:t-1})$ by Gaussians, where the moments are computed using numerical integration; we can then compute the marginal and conditional of these distributions to perform the time and measurement updates.

There are many methods to compute the Gaussian moments, as we discuss in Section 8.5.1. Here we use a method based on the unscented transform (see Section 8.4.1). Using the unscented transform for the transition and observation models gives the overall method, known as the **unscented Kalman filter** or **UKF**, [JU97; JUDW00], also called the **sigma point filter** [VDMW03].

The main advantage of the UKF over the EKF is that it can be more accurate, and more stable. (Indeed, [JU97; JUDW00] claim the term “unscented” was invented because the method “doesn’t stink”.) In addition, the UKF does not need to compute Jacobians of the observation and dynamics models, so it can be applied to non-differentiable models, or ones with hard constraints. However, the UKF can be slower, since it requires N_z evaluations of the dynamics and observation models. In addition, it has 3 hyper-parameters that need to be set.

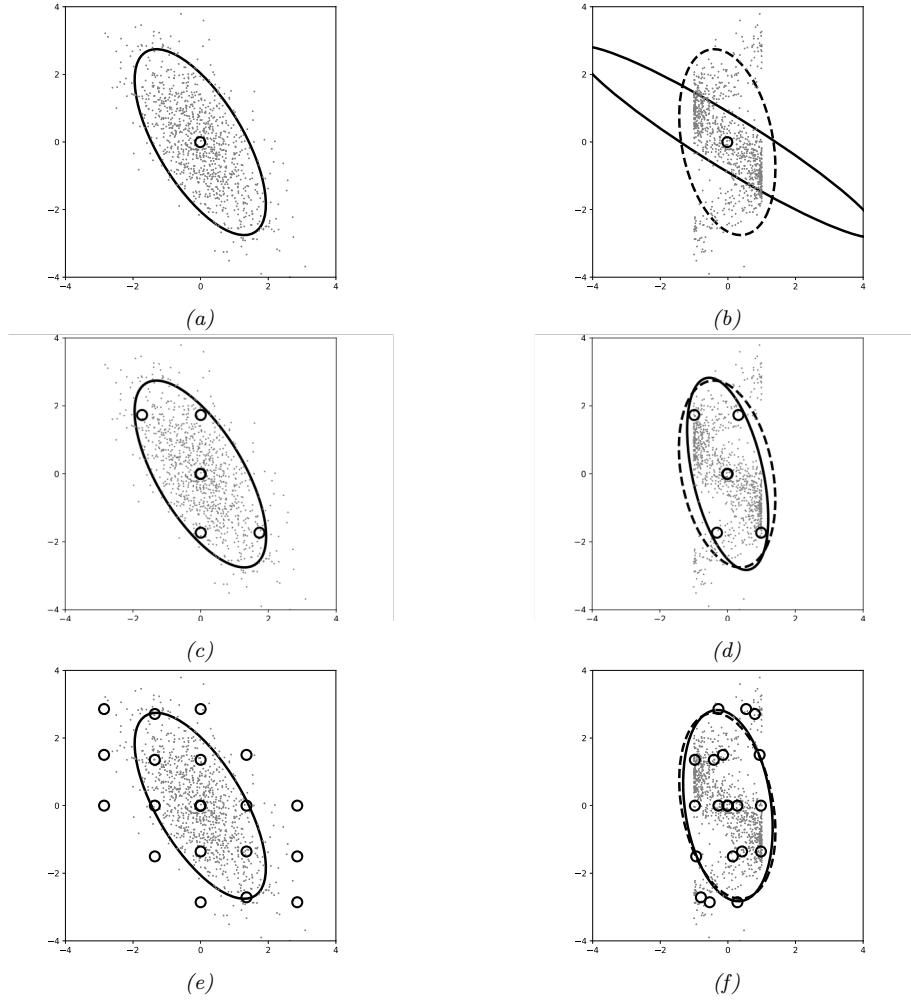


Figure 8.5: Illustration of different ways to approximate the distribution induced by a nonlinear transformation $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$. (a) Data from the source distribution, $\mathcal{D} = \{\mathbf{x}_i \sim p(\mathbf{x})\}$, with Gaussian approximation superimposed. (b) The dots show a Monte Carlo approximation to $p(f(\mathbf{x}))$ derived from $\mathcal{D}' = \{f(\mathbf{x}_i)\}$. The dotted ellipse is a Gaussian approximation to this target distribution, computed from the empirical moments. The solid ellipse is a Taylor transform. (c) Unscented sigma points. (d) Unscented transform. (e) Gauss-Hermite points (order 5). (f) GH transform. Adapted from Figures 5.3–5.4 of [Sar13]. Generated by gaussian_transforms.ipynb.

Algorithm 8.5: Computing sigma points using unscented transform.

- 1 def **SigmaPoints**($\mu, \Sigma; \alpha, \beta, \kappa$) :
 - 2 $n = \text{dimensionality of } \mu$
 - 3 $\lambda = \alpha^2(n + \kappa) - n$
 - 4 Compute a set of $2n + 1$ sigma points:

$$\mathcal{X}_0 = \mu, \quad \mathcal{X}_i = \mu + \sqrt{n + \lambda} [\sqrt{\Sigma}]_{:,i}, \quad \mathcal{X}_{i+n} = \mu - \sqrt{n + \lambda} [\sqrt{\Sigma}]_{:,i}$$
 - 5 Compute a set of $2n + 1$ weights for the mean and covariance:

$$w_0^m = \frac{\lambda}{n + \lambda}, \quad w_0^c = \frac{\lambda}{n + \lambda} + (1 - \alpha^2 + \beta), \quad w_i^m = w_i^c = \frac{1}{2(n + \lambda)}$$
 - 6 Return $(\mathcal{X}_{0:2n}, w_{0:2n}^m, w_{0:2n}^c)$
-

Algorithm 8.6: Unscented approximation to a joint Gaussian distribution.

- 1 def **UnscentedMoments**($\mu, \Sigma, g, \Omega; \alpha, \beta, \kappa$) :
 - 2 $(\mathcal{X}_{0:2n}, w_{0:2n}^m, w_{0:2n}^c) = \text{SigmaPoints}(\mu, \Sigma; \alpha, \beta, \kappa)$
 - 3 $\mathcal{Y}_i = g(\mathcal{X}_i), \quad i = 0 : 2n$
 - 4 $\hat{y} = \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i$
 - 5 $\mathbf{S} = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \mu_U)(\mathcal{Y}_i - \mu_U)^T + \Omega$
 - 6 $\mathbf{C} = \sum_{i=0}^{2n} w_i^c (\mathcal{X}_i - \mu)(\mathcal{Y}_i - \mu_U)^T$
 - 7 Return $(\hat{y}, \mathbf{S}, \mathbf{C})$
-

8.4.1 The unscented transform

Suppose we have two random variables $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$ and $\mathbf{y} = \mathbf{g}(\mathbf{x})$, where $\mathbf{g} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. The unscented transform forms a Gaussian approximation to $p(\mathbf{y})$ using the following process. First we compute a set of $2n + 1$ sigma points, \mathcal{X}_i , and corresponding weights, w_i^m and w_i^c , using Algorithm 8.5, for $i = 0 : 2n$. (The notation $\mathbf{M}_{:,i}$ means the i 'th column of matrix \mathbf{M} , $\sqrt{\Sigma}$ is the matrix square root, so $\sqrt{\Sigma}\sqrt{\Sigma}^T = \Sigma$.) Next we propagate the sigma points through the nonlinear function to get the following $2n + 1$ outputs:

$$\mathcal{Y}_i = \mathbf{g}(\mathcal{X}_i), \quad i = 0 : 2n \tag{8.148}$$

Finally we estimate the mean and covariance of the resulting set of points:

$$\mathbb{E}[\mathbf{g}(\mathbf{x})] \approx \hat{\mathbf{y}} = \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i \tag{8.149}$$

$$\text{Cov}[\mathbf{g}(\mathbf{x})] \approx \mathbf{S}' = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \hat{\mathbf{y}})(\mathcal{Y}_i - \hat{\mathbf{y}})^T \tag{8.150}$$

Now suppose we want to approximate the joint distribution $p(\mathbf{x}, \mathbf{y})$, where $\mathbf{y} = \mathbf{g}(\mathbf{x}) + \mathbf{e}$, and $\mathbf{e} \sim \mathcal{N}(\mathbf{0}, \Omega)$. By defining the augmented function $\tilde{\mathbf{g}}(\mathbf{x}) = (\mathbf{x}, \mathbf{g}(\mathbf{x}))$, and applying the above

procedure (and adding extra noise), we get

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu} \\ \hat{\mathbf{y}} \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{S} \end{pmatrix} \right) \quad (8.151)$$

where the parameters are computed using Algorithm 8.6.

The sigma points and their weights depend on three hyperparameters, α , β , and κ , which determine the spread of the sigma points around the mean. A typical recommended setting for these is $\alpha = 10^{-3}$, $\kappa = 1$, $\beta = 2$ [Bit16].

In Figure 8.5(a-b), we show the linearized Taylor transform discussed in Section 8.3.1 applied to a nonlinear function. In Figure 8.5(c-d), we show the corresponding unscented transform, which we can see is more accurate. In fact, the unscented transform (which uses $2n + 1$ sigma points) is a third-order method in the sense that the mean of \mathbf{y} is exact for polynomials up to order 3. However the covariance is only exact for linear functions (first order polynomials), because the square of a second order polynomial is already order 4. However, the UT idea can be extended to order 5 using $2n^2 + 1$ sigma points [MS67]; this can capture covariance terms exactly for quadratic functions. We discuss even more accurate approximations, based on numerical integration methods, in Section 8.5.1.4.

8.4.2 The unscented Kalman filter (UKF)

The UKF applies the unscented transform twice, once to approximate passing through the system model \mathbf{f} , and once to approximate passing through the measurement model \mathbf{h} . By analogy to Section 8.2.2.5, we can derive the UKF algorithm as follows:

$$(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \text{UnscentedMoments}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, \mathbf{f}(\cdot, \mathbf{u}_t), \mathbf{Q}_t) \quad (8.152)$$

$$(\hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t) = \text{UnscentedMoments}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \mathbf{h}(\cdot, \mathbf{u}_t), \mathbf{R}_t) \quad (8.153)$$

$$(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \text{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t, \mathbf{y}_t) \quad (8.154)$$

See [SS23, Sec 8.8] for more details.

In Figure 8.4c, we illustrate the UKF algorithm (with $\alpha = 1$, $\beta = 0$, $\kappa = 2$) applied to the 2d nonlinear tracking problem from Section 8.3.2.3.

8.4.3 The unscented Kalman smoother (UKS)

The **unscented Kalman smoother**, also called the **unscented RTS smoother** [Sar08], is a simple modification of the usual Kalman smoothing method, where we approximate the nonlinearity by the unscented transform. The key insight is to notice that the reverse Kalman gain matrix \mathbf{J}_t in Equation (8.80) can be defined in terms of the predicted covariance and cross covariance, both of which can be estimated using the UT. Once we have computed this, we can use the RTS equations in the usual way. See [SS23, Sec 14.4] for the details.

An interesting application of unscented Kalman smoothing was its use by the UK government as part of its COVID-19 contact tracing app [Lov+20; BCH20]. The app used the UKS to estimate the distance between (anonymized) people based on bluetooth signal strength between their mobile phones; the distance was then combined with other signals, such as contact duration and infectiousness level of the index case, to estimate the risk of transmission. (See also [MKS21] for a way to learn the risk score.)

8.5 Other variants of the Kalman filter

In this section, we briefly mention some other variants of Kalman filtering. For a more extensive review, see [Sar13; SS23; Li+17e].

8.5.1 General Gaussian filtering

This section is co-authored with Peter Chang.

Let $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{y}|\mathbf{h}_{\boldsymbol{\mu}}(\mathbf{z}), \boldsymbol{\Omega})$ for some function $\mathbf{h}_{\boldsymbol{\mu}}$. Let $p(\mathbf{z}, \mathbf{y}) = p(\mathbf{z})p(\mathbf{y}|\mathbf{z})$ be the exact joint distribution. The best Gaussian approximation to the joint can be computed by solving

$$q(\mathbf{z}, \mathbf{y}) = \underset{q \in \mathcal{N}}{\operatorname{argmin}} D_{\text{KL}}(p(\mathbf{z}, \mathbf{y}) \parallel q(\mathbf{z}, \mathbf{y})) \quad (8.155)$$

As we explain in Section 5.1.4.2, this can be obtained by **moment matching**, i.e.,

$$q(\mathbf{z}, \mathbf{y}) = \mathcal{N}\left(\begin{pmatrix} \mathbf{z} \\ \mathbf{y} \end{pmatrix} \mid \begin{pmatrix} \boldsymbol{\mu} \\ \hat{\mathbf{y}} \end{pmatrix}, \begin{pmatrix} \boldsymbol{\Sigma} & \mathbf{C} \\ \mathbf{C}^T & \mathbf{S} \end{pmatrix}\right) \quad (8.156)$$

where

$$\hat{\mathbf{y}} = \mathbb{E}[\mathbf{y}] = \int \mathbf{h}_{\boldsymbol{\mu}}(\mathbf{z}) \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.157)$$

$$\mathbf{S} = \mathbb{V}[\mathbf{y}] = \boldsymbol{\Omega} + \int (\mathbf{h}_{\boldsymbol{\mu}}(\mathbf{z}) - \hat{\mathbf{y}})(\mathbf{h}_{\boldsymbol{\mu}}(\mathbf{z}) - \hat{\mathbf{y}})^T \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.158)$$

$$\mathbf{C} = \operatorname{Cov}[\mathbf{z}, \mathbf{y}] = \int (\mathbf{z} - \boldsymbol{\mu})(\mathbf{h}_{\boldsymbol{\mu}}(\mathbf{z}) - \hat{\mathbf{y}})^T \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.159)$$

We can use the above Gaussian approximation either for the time update (i.e., going from $p(\mathbf{z}_{t-1}|\mathbf{y}_{1:t-1})$ to $p(\mathbf{z}_t|\mathbf{y}_{1:t-1})$ via $p(\mathbf{z}_{t-1}, \mathbf{z}_t|\mathbf{y}_{1:t-1})$), or for the measurement update, (i.e., going from $p(\mathbf{z}_t|\mathbf{y}_{1:t-1})$ to $p(\mathbf{z}_t|\mathbf{y}_{1:t})$ via $p(\mathbf{z}_t, \mathbf{y}_t|\mathbf{y}_{1:t-1})$). For example, if the prior from the time update is $p(\mathbf{z}_t) = \mathcal{N}(\mathbf{z}_t|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1})$, then the measurement update becomes

$$\mathbf{K}_t = \mathbf{C}_t \mathbf{S}_t^{-1} \quad (8.160)$$

$$\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t(\mathbf{y}_t - \hat{\mathbf{y}}_t) \quad (8.161)$$

$$\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^T \quad (8.162)$$

The resulting method is called **general Gaussian filtering** or **GGF** [IX00; Wu+06].

8.5.1.1 Statistical linear regression

An alternative perspective on the above method is that we are approximating the likelihood by $q(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\mathbf{z} + \mathbf{b}, \boldsymbol{\Omega})$, where we define

$$\begin{aligned} \mathbf{A} &= \mathbf{C}^T \boldsymbol{\Sigma}^{-1} \\ \mathbf{b} &= \hat{\mathbf{y}} - \mathbf{A}\boldsymbol{\mu} \\ \boldsymbol{\Omega} &= \mathbf{S} - \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^T \end{aligned} \quad (8.163)$$

This is called **statistical linear regression** or **SLR** [LBS01; AHE07], and ensures that we minimize

$$\mathcal{L}(\mathbf{A}, \mathbf{b}, \boldsymbol{\Omega}) = \mathbb{E}_{\mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})} [D_{\text{KL}}(p(\mathbf{y}|\mathbf{z}) \parallel q(\mathbf{y}|\mathbf{z}; \mathbf{A}, \mathbf{b}, \boldsymbol{\Omega}))] \quad (8.164)$$

For the proof, see [GF+15; Kam+22].

Equivalently, one can show that the above parameters minimize the following mean squared error

$$\mathcal{L}(\mathbf{A}, \mathbf{b}) = \mathbb{E} [(\mathbf{y} - \mathbf{Ax} - \mathbf{b})^\top (\mathbf{y} - \mathbf{Ax} - \mathbf{b})] \quad (8.165)$$

with $\boldsymbol{\Omega}$ given by the residual noise

$$\boldsymbol{\Omega} = \mathbb{E} [(\mathbf{y} - \mathbf{Ax} - \mathbf{b})(\mathbf{y} - \mathbf{Ax} - \mathbf{b})^\top] \quad (8.166)$$

See [SS23, Sec 9.4] for the proof.

Note that although SLR results in a linear model, it is different than the Taylor series approximation of Section 8.3.1, since the linearization is chosen to be optimal wrt a distribution of points (averaged over $\mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$), instead of just being optimal at a single point $\boldsymbol{\mu}$.

8.5.1.2 Approximating the moments

To implement GGF, we need a way to compute $\hat{\mathbf{y}}$, \mathbf{S} and \mathbf{C} . To help with this, we define two functions to compute Gaussian first and second moments:

$$g_e(\mathbf{f}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \triangleq \int \mathbf{f}(\mathbf{z}) \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.167)$$

$$g_c(\mathbf{f}, \mathbf{g}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) \triangleq \int (\mathbf{f}(\mathbf{z}) - \bar{\mathbf{f}})(\mathbf{g}(\mathbf{z}) - \bar{\mathbf{g}})^\top \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.168)$$

where $\bar{\mathbf{f}} = g_e(\mathbf{f}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\bar{\mathbf{g}} = g_e(\mathbf{g}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$. There are several ways to compute these integrals, as we discuss below.

8.5.1.3 Approximation based on linearization

The simplest approach to approximating the moments is to linearize the functions \mathbf{f} and \mathbf{g} around $\boldsymbol{\mu}$, which yields the following (see Section 8.3.1):

$$\hat{\mathbf{f}}(\mathbf{z}) = \boldsymbol{\mu} + \mathbf{F}(\mathbf{z} - \boldsymbol{\mu}) \quad (8.169)$$

$$\hat{\mathbf{g}}(\mathbf{z}) = \boldsymbol{\mu} + \mathbf{G}(\mathbf{z} - \boldsymbol{\mu}) \quad (8.170)$$

where \mathbf{F} and \mathbf{G} are the Jacobians of f and g . Thus we get the following implementation of the moment functions:

$$g_e(\hat{\mathbf{f}}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathbb{E}[\boldsymbol{\mu} + \mathbf{F}(z - \boldsymbol{\mu})] = \boldsymbol{\mu} \quad (8.171)$$

$$g_c(\hat{\mathbf{f}}, \hat{\mathbf{g}}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathbb{E}[(\hat{\mathbf{f}}(z) - \bar{\mathbf{f}})(\hat{\mathbf{g}}(z) - \bar{\mathbf{g}})^\top] \quad (8.172)$$

$$= \mathbb{E}[\hat{\mathbf{f}}(z)\hat{\mathbf{g}}(z)^\top + \bar{\mathbf{f}}\bar{\mathbf{g}}^\top - \hat{\mathbf{f}}(z)\bar{\mathbf{g}}^\top - \bar{\mathbf{f}}\hat{\mathbf{g}}(z)^\top] \quad (8.173)$$

$$= \mathbb{E}[(\boldsymbol{\mu} + \mathbf{F}(z - \boldsymbol{\mu}))(\boldsymbol{\mu} + \mathbf{G}(z - \boldsymbol{\mu}))^\top + \boldsymbol{\mu}\boldsymbol{\mu}^\top - \boldsymbol{\mu}\boldsymbol{\mu}^\top - \boldsymbol{\mu}\boldsymbol{\mu}^\top] \quad (8.174)$$

$$= \mathbb{E}[\boldsymbol{\mu}\boldsymbol{\mu}^\top + \mathbf{F}(z - \boldsymbol{\mu})(z - \boldsymbol{\mu})^\top \mathbf{G}^\top + \mathbf{F}(z - \boldsymbol{\mu})\boldsymbol{\mu}^\top + \boldsymbol{\mu}(z - \boldsymbol{\mu})^\top \mathbf{G}^\top - \boldsymbol{\mu}\boldsymbol{\mu}^\top] \quad (8.175)$$

$$= \mathbf{F}\mathbb{E}[(z - \boldsymbol{\mu})(z - \boldsymbol{\mu})^\top] \mathbf{G}^\top = \mathbf{F}\boldsymbol{\Sigma}\mathbf{G}^\top \quad (8.176)$$

Using this inside the GGF is equivalent to the EKF in Section 8.3.2. However, this approach can lead to large errors and sometimes divergence of the filter [IX00; VDMW03].

8.5.1.4 Approximation based on Gaussian quadrature

Since we are computing integrals wrt a Gaussian measure, we can use **Gaussian quadrature** methods of the following form:

$$\int \mathbf{h}(z)\mathcal{N}(z|\boldsymbol{\mu}, \boldsymbol{\Sigma})dz \approx \sum_{k=1}^K w^k \mathbf{h}(z^k) \quad (8.177)$$

for a suitable set of evaluation points z^k (sometimes called **sigma points**) and weights w^k . (Note that one-dimensional integrals are called **quadratures**, and multi-dimensional integrals are called **cubatures**.)

One way to compute the sigma points is to use the unscented transform described in Section 8.4.1. Using this inside the GGF is equivalent to the UKF in Section 8.4.2.

Alternatively, we can use **spherical cubature integration**, which gives rise to the **cubature Kalman filter** or **CKF** [AH09]. This turns out (see [SS23, Sec 8.7]) to be a special case of the UKF, with $2n_z + 1$ sigma points, and hyperparameter values of $\alpha = 1$ and $\beta = 0$ (with κ left free).

A more accurate approximation uses **Gauss-Hermite integration**, which allows the user to select more sigma points. In particular, an order p approximation will be exact for polynomials of order up to $2p - 1$. See [SS23, Sec 8.3] for details, and Figure 8.5(e-f) for an illustration. However, this comes at a price: the number of sigma points is now p^n . Using Gauss-Hermite integration for GGF gives rise to the **Gauss-Hermite Kalman filter** or **GHKF** [IX00], also known as the **quadrature Kalman filter** or **QKF** [AHE07].

8.5.1.5 Approximation based on Monte Carlo integration

We can also approximate the integrals with Monte Carlo (see Section 11.2). Note, however, that this is not the same as particle filtering (Section 13.2), which approximates the conditional $p(z_t|y_{1:t})$ rather than the joint $p(z_t, y_t|y_{1:t-1})$ (see Section 8.6.1 for discussion of this difference).

8.5.2 Conditional moment Gaussian filtering

We can go beyond the Gaussian likelihood assumption by approximating the actual likelihood by a linear Gaussian model, as proposed in [TGFS18]. The only requirement is that we can compute the first and second **conditional moments** of the likelihood:

$$\mathbf{h}_\mu(\mathbf{z}) = \mathbb{E}[\mathbf{y}|\mathbf{z}] = \int \mathbf{y} p(\mathbf{y}|\mathbf{z}) d\mathbf{y} \quad (8.178)$$

$$\mathbf{h}_\Sigma(\mathbf{z}) = \text{Cov}[\mathbf{y}|\mathbf{z}] = \int (\mathbf{y} - \mathbf{h}_\mu(\mathbf{z}))(\mathbf{y} - \mathbf{h}_\mu(\mathbf{z}))^\top p(\mathbf{y}|\mathbf{z}) d\mathbf{y} \quad (8.179)$$

Note that these integrals may be wrt a non-Gaussian measure $p(\mathbf{y}|\mathbf{z})$. Also, \mathbf{y} may be discrete, in which case these integrals become sums.

Next we compute the unconditional moments. By the law of iterated expectations we have

$$\hat{\mathbf{y}} = \mathbb{E}[\mathbf{y}] = \mathbb{E}[\mathbb{E}[\mathbf{y}|\mathbf{z}]] = \int \mathbf{h}_\mu(\mathbf{z}) \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} = g_e(\mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (8.180)$$

Similarly

$$\mathbf{C} = \text{Cov}[\mathbf{z}, \mathbf{y}] = \mathbb{E}[\mathbb{E}[(\mathbf{z} - \boldsymbol{\mu})(\mathbf{y} - \hat{\mathbf{y}})|\mathbf{z}]] = \mathbb{E}[(\mathbf{z} - \boldsymbol{\mu})(\mathbf{h}_\mu(\mathbf{z}) - \hat{\mathbf{y}})] \quad (8.181)$$

$$= \int (\mathbf{z} - \boldsymbol{\mu})(\mathbf{h}_\mu(\mathbf{z}) - \hat{\mathbf{y}})^\top \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} = g_c(\mathbf{z}, \mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (8.182)$$

Finally

$$\mathbf{S} = \mathbb{V}[\mathbf{y}] = \mathbb{E}[\mathbb{V}[\mathbf{y}|\mathbf{z}]] + \mathbb{V}[\mathbb{E}[\mathbf{y}|\mathbf{z}]] \quad (8.183)$$

$$= \int \mathbf{h}_\Sigma(\mathbf{z}) \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} + \int (\mathbf{h}_\mu(\mathbf{z}) - \hat{\mathbf{y}})(\mathbf{h}_\mu(\mathbf{z}) - \hat{\mathbf{y}})^\top \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{z} \quad (8.184)$$

$$= g_e(\mathbf{h}_\Sigma(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma}) + g_c(\mathbf{h}_\mu(\mathbf{z}), \mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (8.185)$$

Note that the equation for $\hat{\mathbf{y}}$ is the same in Equation (8.157) and Equation (8.180), and the equation for \mathbf{C} is the same in Equation (8.159) and Equation (8.182). Furthermore, if $\mathbf{h}_\Sigma(\mathbf{z}) = \boldsymbol{\Omega}$, then the equation for \mathbf{S} is the same in Equation (8.158) and Equation (8.185).

We can approximate the unconditional moments using linearization or numerical integration. We can then plug them into the GGF algorithm. We call this **conditional moments Gaussian filtering** or **CMGF**.

We can use CMGF to perform approximate inference in SSMs with Poisson likelihoods. For example, if $p(y|z) = \text{Poisson}(y|ce^z)$, we have

$$\mathbf{h}_\mu(z) = \mathbf{h}_\Sigma(z) = ce^z \quad (8.186)$$

This method can be used to perform (extended) Kalman filtering with more general exponential family likelihoods, as described in [TGFS18; Oll18]. For example, suppose we have a categorical likelihood:

$$p(y_t|\mathbf{z}_t) = \text{Cat}(y_t|\mathbf{p}_t) = \text{Cat}(y_t|\text{softmax}(\boldsymbol{\eta}_t)) = \text{Cat}(y_t|\text{softmax}(h(\mathbf{z}_t))) \quad (8.187)$$

where $\eta_t = h(\mathbf{z}_t)$ are the predicted logits. Then the conditional mean and covariance are given by

$$\mathbf{h}_\mu(\mathbf{z}_t) = \mathbf{p}_t = \text{softmax}(h(\mathbf{z}_t)), \quad \mathbf{h}_\Sigma(\mathbf{z}_t) = \text{diag}(\mathbf{p}_t) - \mathbf{p}_t \mathbf{p}_t^\top \quad (8.188)$$

(We can drop one of the classes from the vector \mathbf{p}_t to ensure the covariance is full rank.) This approach can be used for online inference in neural network classifiers [CMJ22], as well as Gaussian process classifiers [GFTS19] and recommender systems [GU16; GUK21]. We can also use this method as a proposal distribution inside of a particle filtering algorithm (Section 13.2), as discussed in [Hos+20b].

8.5.3 Iterated filters and smoothers

The GGF method in Section 8.5.1, and the CMGF method in Section 8.5.2, both require computing moments wrt the predictive distribution $\mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1})$ before performing the measurement update. It is possible to do one step of GGF to compute the posterior given the new observation, $\mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$, and then to use this revised posterior to compute new moments in an iterated fashion. This is called **iterated posterior linearization filter** or **IPLF** [GF+15]. (This is similar to the iterated EKF which we discussed in Section 8.3.2.2.) See Algorithm 8.7 for the pseudocode, and [SS23, Sec 10.4] for more details.

Algorithm 8.7: Iterated conditional moments Gaussian filter.

```

1 def Iterated-CMGF( $f, \mathbf{Q}, \mathbf{h}_\mu, \mathbf{h}_\Sigma, \mathbf{y}_{1:T}, \boldsymbol{\mu}_{0|0}, \boldsymbol{\Sigma}_{0|0}, J, g_e, g_c$ ) :
2   foreach  $t = 1 : T$  do
3     Predict step:
4      $(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, -) = \text{CondMoments}(\boldsymbol{\mu}_{t-1|t-1}, \boldsymbol{\Sigma}_{t-1|t-1}, f, \mathbf{Q}, g_e, g_c)$ 
5     Update step:
6      $\boldsymbol{\mu}_{t|t} = \boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1}$ 
7     foreach  $j = 1 : J$  do
8        $(\hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t) = \text{CondMoments}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \mathbf{h}_\mu, \mathbf{h}_\Sigma, g_e, g_c)$ 
9        $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \ell_t) = \text{GaussCondition}(\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\Sigma}_{t|t-1}, \hat{\mathbf{y}}_t, \mathbf{S}_t, \mathbf{C}_t, \mathbf{y}_t)$ 
10  Return  $(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})_{t=1}^T$ 
11 def CondMoments( $\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{h}_\mu, \mathbf{h}_\Sigma, g_e, g_c$ ) :
12    $\hat{\mathbf{y}} = g_e(\mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma})$ 
13    $\mathbf{S} = g_e(\mathbf{h}_\Sigma(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma}) + g_c(\mathbf{h}_\mu(\mathbf{z}), \mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma})$ 
14    $\mathbf{C} = g_c(\mathbf{z}, \mathbf{h}_\mu(\mathbf{z}), \boldsymbol{\mu}, \boldsymbol{\Sigma})$ 
15  Return  $(\hat{\mathbf{y}}, \mathbf{S}, \mathbf{C})$ 

```

In a similar way, we can derive the **iterated posterior linearization smoother** or **IPLS** [GFSS17]. This is similar to the iterated EKS which we discussed in Section 8.3.3.

Unfortunately the IPLF and IPLS can diverge. A more robust version of IPLF, that uses line search to perform damped (partial) updates, is presented in [Rai+18b]. Similarly, a more robust version of IPLS, that uses line search and Levenberg-Marquardt to update the parameters, is presented in [Lin+21c].

Various extensions of the above methods have been proposed. For example, in [HPR19] they extend IPLS to belief propagation in Forney factor graphs (Section 4.6.1.2), which enables the method to be applied to a large class of graphical models beyond SSMs. In particular, they give a general linearization formulation (including explicit message update rules) for nonlinear approximate Gaussian BP (Section 9.4.3) where the linearization can be Jacobian-based (“EKF-style”), statistical (moment matching), or anything else. They also show how any such linearization method can benefit from iterations.

In [Kam+22], they present a method based on approximate expectation propagation (Section 10.7), that is very similar to IPLS, except that the distributions that are used to compute the SLR terms, needed to compute the Gaussian messages, are different. In particular, rather than using the smoothed posterior from the last iteration, it uses the “cavity” distribution, which is the current posterior minus the incoming message that was sent at the last iteration, similar to Section 8.2.4.4. The advantage of this is that the outgoing message does not double count the evidence. The disadvantage is that this may be numerically unstable.

In [WSS21], they propose a variety of “**Bayes-Newton**” methods for approximately computing Gaussian posteriors to probabilistic models with nonlinear and/or non-Gaussian likelihoods. This generalizes all of the above methods, and can be applied to SSMs and GPs.

8.5.4 Ensemble Kalman filter

The **ensemble Kalman filter (EnKF)** is a technique developed in the geoscience (meteorology) community to perform approximate online inference in large nonlinear systems. In particular, it is mostly used for problems where the hidden state represents an unknown physical quantity (e.g., temperature or pressure) at each point on a spatial grid, and the measurements are sparse and spatially localized. Combining this information over space and time is called **data assimilation**.

The canonical reference is [Eve09], but a more accessible tutorial (using the same Bayesian signal processing approach we adopt in this chapter) is in [Rot+17].

The key idea is to represent the belief state $p(\mathbf{z}_t | \mathbf{y}_{1:t})$ by a finite number of samples $\mathbf{Z}_{t|t} = \{\mathbf{z}_{t|t}^s : s = 1 : N_s\}$, where each $\mathbf{z}_{t|t}^s \in \mathbb{R}^{N_z}$. In contrast to particle filtering (Section 13.2), the samples are updated in a manner that closely resembles the Kalman filter, so there is no importance sampling or resampling step. The downside is that the posterior does not converge to the true Bayesian posterior even as $N_s \rightarrow \infty$ [LGMT11], except in the linear-Gaussian case. However, sometimes the performance of EnKF can be better for small number of samples (although this depends of course on the PF proposal distribution).

The posterior mean and covariance can be derived from the ensemble of samples as follows:

$$\tilde{\mathbf{z}}_{t|t} = \frac{1}{N_s} \sum_{s=1}^{N_s} \mathbf{z}_{t|t}^s = \frac{1}{N_s} \mathbf{Z}_{t|t} \mathbf{1} \quad (8.189)$$

$$\tilde{\Sigma}_{t|t} = \frac{1}{N_s - 1} \sum_{s=1}^{N_s} (\mathbf{z}_t^s - \tilde{\mathbf{z}}_{t|t})(\mathbf{z}_t^s - \tilde{\mathbf{z}}_{t|t})^\top = \frac{1}{N_s - 1} \tilde{\mathbf{Z}}_{t|t} \tilde{\mathbf{Z}}_{t|t}^\top \quad (8.190)$$

where $\tilde{\mathbf{Z}}_{t|t} = \mathbf{Z}_{t|t} - \tilde{\mathbf{z}}_{t|t} \mathbf{1}^\top = \mathbf{Z}_{t|t} (\mathbf{I}_{N_s} - \frac{1}{N_s} \mathbf{1} \mathbf{1}^\top)$.

We update the samples as follows. For the time update, we first draw N_s system noise variables $\mathbf{q}_t^s \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_t)$, and then we pass these, and the previous state estimate, through the dynamics

model to get the one-step-ahead state predictions, $\mathbf{z}_{t|t-1}^s = \mathbf{f}(\mathbf{z}_{t-1|t-1}^s, \mathbf{q}_t^s)$, from which we get $\mathbf{Z}_{t|t-1} = \{\mathbf{z}_{t|t-1}^s\}$, which has size $N_z \times N_s$. Next we draw N_s observation noise variables $\mathbf{r}_t^s \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t)$, and use them to compute the one-step-ahead observation predictions, $\mathbf{y}_{t|t-1}^s = \mathbf{h}(\mathbf{z}_{t|t-1}^s, \mathbf{r}_t^s)$ and $\mathbf{Y}_{t|t-1} = \{\mathbf{y}_{t|t-1}^s\}$, which has size $N_y \times N_s$. Finally we compute the measurement update using

$$\tilde{\mathbf{Z}}_{t|t} = \mathbf{Z}_{t|t-1} + \tilde{\mathbf{K}}_t (\mathbf{y}_t \mathbf{1}^\top - \mathbf{Y}_{t|t-1}) \quad (8.191)$$

which is the analog of Equation (8.29).

We now discuss how to compute $\tilde{\mathbf{K}}_t$, which is the analog of the Kalman gain matrix in Equation (8.28). First note that we can write the exact Kalman gain matrix (in the linear-Gaussian case) as $\mathbf{K}_t = \mathbf{\Sigma}_{t|t-1} \mathbf{H}^\top \mathbf{S}_t^{-1} = \mathbf{C}_t \mathbf{S}_t^{-1}$, where \mathbf{S}_t is the covariance of the predictive distribution for the observation vector at time t , and \mathbf{C}_t is the cross-covariance of the joint predictive distribution for the next state and next observation. In the EnKF, we approximate \mathbf{S}_t and \mathbf{C}_t empirically as follows. First we compute the anomalies

$$\tilde{\mathbf{Z}}_{t|t-1} = \mathbf{Z}_{t|t-1} - \tilde{\mathbf{z}}_{t|t-1} \mathbf{1}^\top, \quad \tilde{\mathbf{Y}}_{t|t-1} = \mathbf{Y}_{t|t-1} - \tilde{\mathbf{y}}_{t|t-1} \mathbf{1}^\top \quad (8.192)$$

Then we compute the sample covariance matrices

$$\tilde{\mathbf{C}}_t = \frac{1}{N_s - 1} \tilde{\mathbf{Z}}_{t|t-1} \tilde{\mathbf{Y}}_{t|t-1}^\top, \quad \tilde{\mathbf{S}}_t = \frac{1}{N_s - 1} \tilde{\mathbf{Y}}_{t|t-1} \tilde{\mathbf{Z}}_{t|t-1}^\top \quad (8.193)$$

Finally we compute

$$\tilde{\mathbf{K}}_t = \tilde{\mathbf{C}}_t \tilde{\mathbf{S}}_t^{-1} \quad (8.194)$$

which has the same form as a multivariate least squares problem. For models with additive noise, we can reduce the variance of this procedure by eliminating the sampling of the predicted observations. Thus we replace $\tilde{\mathbf{Y}}_{t|t-1}$ with its deterministic version, $\tilde{\mathbf{O}}_{t|t-1} = \mathbf{H} \tilde{\mathbf{Z}}_{t|t-1}$ (assuming a linear observation model for notational simplicity). We then use $\tilde{\mathbf{C}}_t = \frac{1}{N_s - 1} \tilde{\mathbf{Z}}_{t|t-1} \tilde{\mathbf{O}}_{t|t-1}^\top$, and $\tilde{\mathbf{S}}_t = \frac{1}{N_s - 1} \tilde{\mathbf{O}}_{t|t-1} \tilde{\mathbf{O}}_{t|t-1}^\top + \mathbf{R}_t$. (It is also possible to eliminate the sampling for the latent states, by using the **ensemble square root filter** [Tip+03], although this may be less robust.)

We now compare the computational complexity to the KF algorithm. Recall that N_z is the number of latent dimensions, N_y is the number of observed dimensions, and N_s is the number of samples. We will assume $N_z > N_s > N_y$, as occurs in most geospatial problems. The EnKF time update takes $O(N_z^2 N_s)$ time to propagate the samples through the model (assuming that \mathbf{f} is a linear model), whereas the KF takes $O(N_z^3)$ time to compute $\mathbf{F}_t \mathbf{\Sigma}_{t-1} \mathbf{F}_t + \mathbf{Q}_t$. If the transition matrix is sparse, the EnKF time reduces to $O(N_z N_s)$ and the EKF time reduces to (N_z^2) . The EnKF measurement update takes $O(N_z N_y N_s)$ time to compute $\tilde{\mathbf{C}}_t$, $O(N_y^2 N_s)$ time to compute $\tilde{\mathbf{S}}_t$, $O(N_y^3)$ time to compute $\tilde{\mathbf{S}}_t^{-1}$, $O(N_z N_y^2)$ to compute $\tilde{\mathbf{K}}_t$, and $O(N_z N_y N_s)$ time to compute $\mathbf{Z}_{t|t}$, for a total of $O(N_z N_y N_s + N_s N_y^2)$, where we have dropped terms that don't depend on N_z for notational simplicity. By contrast, in the EKF, the measurement update takes $O(N_z^2 N_y)$ to compute \mathbf{C}_t , $O(N_z^2 N_y)$ to compute \mathbf{S}_t , $O(N_y^3)$ to compute \mathbf{S}_t^{-1} , $O(N_z N_y^2)$ to compute \mathbf{K}_t , and $O(N_z N_y)$ to compute $\boldsymbol{\mu}_{t|t}$, for a total of $O(N_z^2 N_y + N_z N_y^2)$. In summary, EnKF is $O(N_z N_s)$, but for EKF is $O(N_z^2)$.

Unfortunately, if N_s is too small, the EnKF can become overconfident, and the filter can diverge. A common heuristic to reduce this is known as **covariance inflation**, in which we replace $\tilde{\mathbf{Z}}_{t|t-1}$ with $\tilde{\mathbf{Z}}_{t|t-1} = \beta(\mathbf{Z}_{t|t-1} - \tilde{\mathbf{z}}_{t|t-1} \mathbf{1}^\top)$ for some fudge factor $\beta > 1$.

Unlike the particle filter, the EnKF is not guaranteed to converge to the correct posterior. However, hybrid PF/EnKF approaches have been developed (see e.g., [LGMT11; FK13b; Rei13]) with better theoretical foundations.

8.5.5 Robust Kalman filters

In practice we often have noise that is non-Gaussian. A common example is when we have clutter, or outliers, in the observation model, or sudden changes in the process model. In this case, we might use the Laplace distribution [Ara+09] or the Student t -distribution [Ara10; RÖG13; Ara+17] as noise models.

[Hua+17b] proposes a variational Bayes (Section 10.3.3) approach, that allows the dynamical prior and the observation model to both be (linear) Student distributions, but where the posterior is approximated at each step using a Gaussian, conditional on the noise scale matrix, which is modeled using an inverse Wishart distribution. An extension of this, to handle mixture distributions, can be found in [Hua+19].

8.5.6 Dual EKF

In this section, we briefly discuss one approach to estimating the parameters of an SSM. In an offline setting, we can use EM, SGD, or Bayesian inference to compute an approximation to $p(\boldsymbol{\theta}|\mathbf{y}_{1:T})$ (see Section 29.8). In the online setting, we want to compute $p(\boldsymbol{\theta}_t|\mathbf{y}_{1:t})$. We can do this by adding the parameters to the state space, possibly with an **artificial dynamics**, $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \epsilon\mathbf{I})$, and then performing joint inference of states and parameters. The latent variables at each step now contain the latent states, \mathbf{z}_t , and the latent parameters, $\boldsymbol{\theta}_t$. One approach to performing approximating inference in such a model is to use the **dual EKF**, in which one EKF performs state estimation and the other EKF performs parameter estimation [WN01].

8.5.7 Normalizing flow KFs

Normalizing flows, as discussed in Chapter 23, are a kind of deep generative model with a tractable exact likelihood. These can be used to “upgrade” the observation model of a linear Gaussian SSM, while still retaining tractable exact Gaussian inference, as shown in [Béz+20]. In particular, instead of observing $\mathbf{y}_t = \mathbf{H}_t \mathbf{z}_t + \mathbf{r}_t$, where $\mathbf{r}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_t)$, we observe $\mathbf{y}_t = f_t(\mathbf{h}_t)$, where $\mathbf{h}_t = \mathbf{H}_t \mathbf{z}_t + \mathbf{r}_t$, and $f_t : \mathbb{R}^{N_y} \rightarrow \mathbb{R}^{N_y}$ is an invertible function with a tractable Jacobian. In this case, the exact posterior, $p(\mathbf{z}_t|\mathbf{y}_{1:t})$, is given by the usual Kalman filter equations applied to $\mathbf{h}_t = f_t^{-1}(\mathbf{y}_t)$ instead of \mathbf{y}_t , which we denote by $p_{\text{LGSSM}}(\mathbf{z}_t|\mathbf{h}_{1:t})$.

This result can be shown by induction. First note that, by the change of variable formula, $p(\mathbf{y}_t|\mathbf{z}_t) = p(\mathbf{h}_t|\mathbf{z}_t)Df_t^{-1}(\mathbf{y}_t)$, where $Dg(\mathbf{a}) \triangleq |\det \text{Jac}(g)(\mathbf{a})|$. By induction, we have $p(\mathbf{z}_{t-1}|\mathbf{y}_{1:t-1}) = p_{\text{LGSSM}}(\mathbf{z}_{t-1}|\mathbf{h}_{1:t-1})$, and hence by the linear Gaussian assumptions for the dynamics model, $p(\mathbf{z}_t|\mathbf{y}_{1:t-1}) = p_{\text{LGSSM}}(\mathbf{z}_t|\mathbf{h}_{1:t-1})$. Thus the filtering posterior is given by

$$\begin{aligned} p(\mathbf{z}_t|\mathbf{y}_{1:t}) &= \frac{p(\mathbf{y}_t|\mathbf{z}_t)p(\mathbf{z}_t|\mathbf{y}_{1:t-1})}{\int p(\mathbf{y}_t|\mathbf{z}'_t)p(\mathbf{z}'_t|\mathbf{y}_{1:t-1})d\mathbf{z}'_t} = \frac{Df_t^{-1}(\mathbf{y}_t)p(\mathbf{h}_t|\mathbf{z}_t)p(\mathbf{z}_t|\mathbf{y}_{1:t-1})}{\int Df_t^{-1}(\mathbf{y}_t)p(\mathbf{h}_t|\mathbf{z}'_t)p(\mathbf{z}'_t|\mathbf{y}_{1:t-1})} \\ &= \frac{p(\mathbf{h}_t|\mathbf{z}_t)p_{\text{LGSSM}}(\mathbf{z}_t|\mathbf{h}_{1:t-1})}{\int p(\mathbf{h}_t|\mathbf{z}'_t)p_{\text{LGSSM}}(\mathbf{z}'_t|\mathbf{h}_{1:t-1})} = p_{\text{LGSSM}}(\mathbf{z}_t|\mathbf{h}_{1:t}) \end{aligned}$$

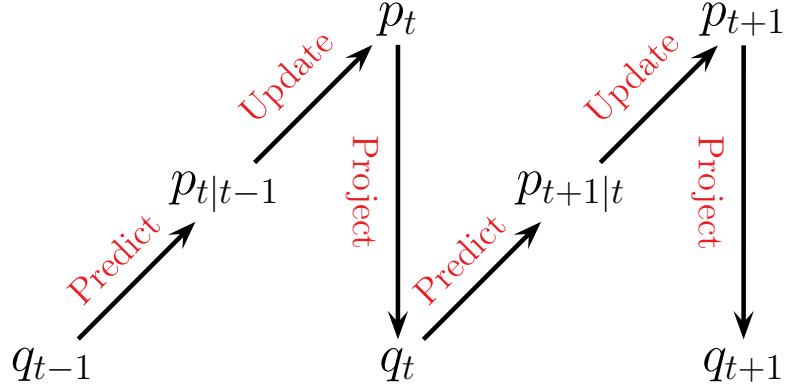


Figure 8.6: Illustration of the predict-update-project cycle of assumed density filtering. $q_t \in \mathcal{Q}$ is a tractable distribution, whereas we may have $p_{t|t-1} \notin \mathcal{Q}$ and $p_t \notin \mathcal{Q}$.

Similar reasoning applies to the smoothing distribution.

8.6 Assumed density filtering

In this section, we discuss **assumed density filtering** or **ADF** [May79]. In this approach, we *assume* the posterior has a specific form (e.g., a Gaussian). At each step, we update the previous posterior with the new likelihood; the result will often not have the desired form (e.g., will no longer be Gaussian), so we project it to the closest approximating distribution of the required type.

In more detail, we assume (by induction) that our prior $q_{t-1}(\mathbf{z}_{t-1}) \approx p(\mathbf{z}_{t-1} | \mathbf{y}_{1:t-1})$ satisfies $q_{t-1} \in \mathcal{Q}$, where \mathcal{Q} is a family of tractable distributions. We can update the prior with the new measurement to get the approximate posterior as follows. First we compute the **one-step-ahead predictive distribution**

$$p_{t|t-1}(\mathbf{z}_t | \mathbf{y}_{1:t-1}) = \int p(\mathbf{z}_t | \mathbf{z}_{t-1}) q_{t-1}(\mathbf{z}_{t-1}) d\mathbf{z}_{t-1} \quad (8.195)$$

Then we update this prior with the likelihood for step t to get the posterior

$$p_t(\mathbf{z}_t | \mathbf{y}_{1:t}) = \frac{1}{Z_t} p(\mathbf{y}_t | \mathbf{z}_t) p_{t|t-1}(\mathbf{z}_t) \quad (8.196)$$

where

$$Z_t = \int p(\mathbf{y}_t | \mathbf{z}_t) p_{t|t-1}(\mathbf{z}_t) d\mathbf{z}_t \quad (8.197)$$

is the normalization constant. Unfortunately, we often find that the resulting posterior is no longer in our tractable family, $p(\mathbf{z}_t) \notin \mathcal{Q}$. So after Bayesian updating we seek the best tractable approximation by computing

$$q_t(\mathbf{z}_t | \mathbf{y}_{1:t}) = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} D_{\text{KL}}(p_t(\mathbf{z}_t | \mathbf{y}_{1:t}) \| q(\mathbf{z}_t)) \quad (8.198)$$

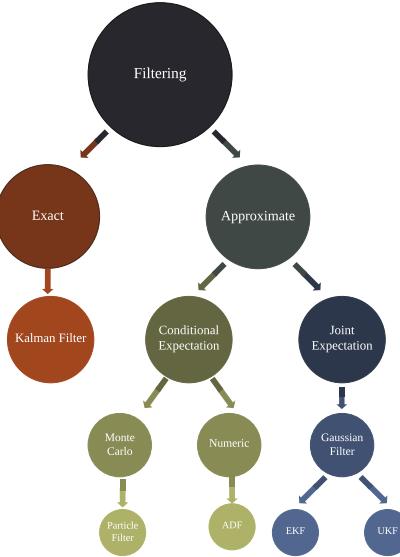


Figure 8.7: A taxonomy of filtering algorithms. Adapted from Figure 2 of [Wütt+16].

This minimizes the Kullback-Leibler divergence from the approximation $q(\mathbf{z}_t)$ to the “exact” posterior $p_t(\mathbf{z}_t)$, and can be thought of as **projecting** p onto the space of tractable distributions. Thus the overall algorithm consists of three steps — predict, update, and project — as sketched in Figure 8.6.

Computing $\min_q D_{\text{KL}}(p \parallel q)$ is known as **moment projection**, since the optimal q should have the same moments as p (see Section 5.1.4.2). So in the Gaussian case, we just need to set the mean and covariance of q_t so they are the same as the mean and covariance of p_t . We will give some examples of this below. By contrast, computing $\min_q D_{\text{KL}}(q \parallel p)$, as in variational inference (Section 10.1), is known as **information projection**, and will result in mode seeking behavior (see Section 5.1.4.1), rather than trying to capture overall moments.

8.6.1 Connection with Gaussian filtering

When \mathcal{Q} is the set of Gaussian distributions, there is a close connection between ADF and Gaussian filtering, which we discussed in Section 8.5.1. GF corresponds to solving the following optimization problem

$$q_{t|t-1}(\mathbf{z}_t, \tilde{\mathbf{y}}_t) = \operatorname{argmin}_{q \in \mathcal{Q}} D_{\text{KL}}(p(\mathbf{z}_t, \tilde{\mathbf{y}}_t | \mathbf{y}_{1:t-1}) \parallel q(\mathbf{z}_t, \tilde{\mathbf{y}}_t | \mathbf{y}_{1:t-1})) \quad (8.199)$$

which can be solved by moment matching (see Section 8.5.1). We then condition this joint distribution on the event $\tilde{\mathbf{y}}_t = \mathbf{y}_t$, where $\tilde{\mathbf{y}}_t$ is the unknown random variable and \mathbf{y}_t is its observed value. This gives $p_t(\mathbf{z}_t | \mathbf{y}_{1:t})$, which is easy to compute, due to the Gaussian assumption. By contrast, in Gaussian ADF, we first compute the (locally) exact posterior $p_t(\mathbf{z}_t | \mathbf{y}_{1:t})$, and then approximate it with $q_t(\mathbf{z}_t | \mathbf{y}_{1:t})$ by projecting into \mathcal{Q} . Thus ADF approximates the conditional $p_t(\mathbf{z}_t | \mathbf{y}_{1:t})$, whereas GF approximates the joint $p_{t|t-1}(\mathbf{z}_t, \tilde{\mathbf{y}}_t | \mathbf{y}_{1:t-1})$, from which we derive $p_t(\mathbf{z}_t | \mathbf{y}_{1:t})$ by conditioning.

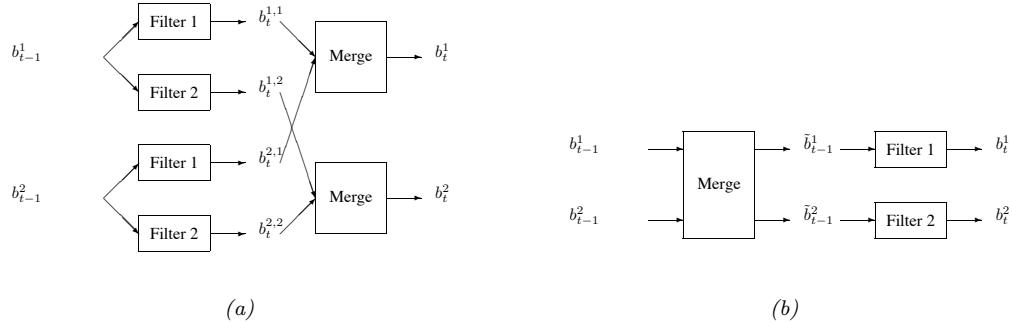


Figure 8.8: ADF for a switching linear dynamical system with 2 discrete states. (a) GPB2 method. (b) IMM method.

ADF is more accurate than GF, since it directly approximates the posterior, but it is more computationally demanding, for reasons explained in [Wüt+16]. However, in [Kam+22] they propose an approximate form of expectation propagation (which is a generalization of ADF) in which the messages are computed using the same local joint Gaussian approximation as used in Gaussian filtering. See Figure 8.7 for a summary of how these different methods relate.

8.6.2 ADF for SLDS (Gaussian sum filter)

In this section, we apply ADF to inference in switching linear dynamical systems (SLDS, Section 29.9), which are a combination of HMM and LDS models. The resulting method is known as the **Gaussian sum filter** (see e.g., [Cro+11; Wil+17]).

A Gaussian sum filter approximates the belief state at each step by a mixture of K Gaussians. This can be implemented by running K Kalman filters in parallel. This is particularly well suited to switching SSMs. We now describe one version of this algorithm, known as the “second order generalized pseudo-Bayes filter” (GPB2) [BSF88]. We assume that the prior belief state b_{t-1} is a mixture of K Gaussians, one per discrete state:

$$b_{t-1}^i \triangleq p(\mathbf{z}_{t-1}, m_{t-1} = i | \mathbf{y}_{1:t-1}) = \pi_{t-1|t-1}^i \mathcal{N}(\mathbf{z}_{t-1} | \boldsymbol{\mu}_{t-1|t-1}^i, \boldsymbol{\Sigma}_{t-1|t-1}^i) \quad (8.200)$$

where $i \in \{1, \dots, K\}$. We then pass this through the K different linear models to get

$$b_t^{ij} \triangleq p(\mathbf{z}_t, m_t = i, m_t = j | \mathbf{y}_{1:t}) = \pi_{t|t}^{ij} \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}^{ij}, \boldsymbol{\Sigma}_{t|t}^{ij}) \quad (8.201)$$

where $\pi_{t|t}^{ij} = \pi_{t-1|t-1}^i A_{ij}$, where $A_{ij} = p(m_t = j | m_{t-1} = i)$. Finally, for each value of j , we collapse the K Gaussian mixtures down to a single mixture to give

$$b_t^j \triangleq p(\mathbf{z}_t, m_t = j | \mathbf{y}_{1:t}) = \pi_{t|t}^j \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_{t|t}^j, \boldsymbol{\Sigma}_{t|t}^j) \quad (8.202)$$

See Figure 8.8a for a sketch.

The optimal way to approximate a mixture of Gaussians with a single Gaussian is given by $q = \arg \min_q D_{\text{KL}}(q \| p)$, where $p(\mathbf{z}) = \sum_k \pi^k \mathcal{N}(\mathbf{z} | \boldsymbol{\mu}^k, \boldsymbol{\Sigma}^k)$ and $q(\mathbf{z}) = \mathcal{N}(\mathbf{z} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$. This can be

solved by moment matching, that is,

$$\boldsymbol{\mu} = \mathbb{E}[\mathbf{z}] = \sum_k \pi^k \boldsymbol{\mu}^k \quad (8.203)$$

$$\boldsymbol{\Sigma} = \text{Cov}[\mathbf{z}] = \sum_k \pi^k \left(\boldsymbol{\Sigma}^k + (\boldsymbol{\mu}^k - \boldsymbol{\mu})(\boldsymbol{\mu}^k - \boldsymbol{\mu})^\top \right) \quad (8.204)$$

In the graphical model literature, this is called **weak marginalization** [Lau92], since it preserves the first two moments. Applying these equations to our model, we can go from b_t^{ij} to b_t^j as follows (where we drop the t subscript for brevity):

$$\pi^j = \sum_i \pi^{ij} \quad (8.205)$$

$$\pi^{j|i} = \frac{\pi^{ij}}{\sum_{j'} \pi^{ij'}} \quad (8.206)$$

$$\boldsymbol{\mu}^j = \sum_i \pi^{j|i} \boldsymbol{\mu}^{ij} \quad (8.207)$$

$$\boldsymbol{\Sigma}^j = \sum_i \pi^{j|i} (\boldsymbol{\Sigma}^{ij} + (\boldsymbol{\mu}^{ij} - \boldsymbol{\mu}^j)(\boldsymbol{\mu}^{ij} - \boldsymbol{\mu}^j)^\top) \quad (8.208)$$

This algorithm requires running K^2 filters at each step. A cheaper alternative, known as **interactive multiple models** or **IMM** [BSF88], can be obtained by first collapsing the prior to a single Gaussian (by moment matching), and then updating it using K different Kalman filters, one per value of m_t . See Figure 8.8b for a sketch.

8.6.3 ADF for online logistic regression

In this section we discuss the application of ADF to online Bayesian parameter inference for a binary logistic regression model, based on [Zoe07]. The overall approach is similar to the online linear regression case (discussed in Section 29.7.2), but approximates the posterior after each update step, which is necessary since the likelihood is not conjugate to the prior.

We assume our model has the following form:

$$p(y_t | \mathbf{x}_t, \mathbf{w}_t) = \text{Ber}(y_t | \sigma(\mathbf{x}_t^\top \mathbf{w}_t)) \quad (8.209)$$

$$p(\mathbf{w}_t | \mathbf{w}_{t-1}) = \mathcal{N}(\mathbf{w}_t | \mathbf{w}_{t-1}, \mathbf{Q}) \quad (8.210)$$

where \mathbf{Q} is the covariance of the process noise, which allows the parameters to change slowly over time. We will assume $\mathbf{Q} = \epsilon \mathbf{I}$; we can also set $\epsilon = 0$, as in the recursive least squares method (Section 29.7.2), if we believe the parameters will not change. See Figure 8.9 for an illustration of the model.

As our approximating family, we will use diagonal Gaussians, for computational efficiency. Thus the prior is the posterior from the previous time step, and has the form

$$p(\mathbf{w}_{t-1} | \mathcal{D}_{1:t-1}) \approx p_{t-1}(\mathbf{w}_{t-1}) = \prod_j \mathcal{N}(w_{t-1}^j | \mu_{t-1|t-1}^j, \tau_{t-1|t-1}^j) \quad (8.211)$$

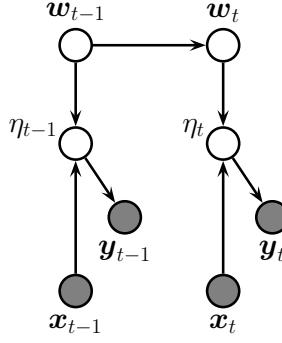


Figure 8.9: A dynamic logistic regression model. w_t are the regression weights at time t , and $\eta_t = w_t^\top \mathbf{x}_t$. Compare to Figure 29.24a.

where $\mu_{t-1|t-1}^j$ and $\tau_{t-1|t-1}^j$ are the posterior mean and variance for parameter j given past data. Now we discuss how to update this prior.

First we compute the one-step-ahead predictive density $p_{t|t-1}(\mathbf{w}_t)$ using the standard linear-Gaussian update, i.e., $\boldsymbol{\mu}_{t|t-1} = \boldsymbol{\mu}_{t-1|t-1}$ and $\boldsymbol{\tau}_{t|t-1} = \boldsymbol{\tau}_{t-1|t-1} + \mathbf{Q}$, where we can set $\mathbf{Q} = 0\mathbf{I}$ if there is no drift.

Now we concentrate on the measurement update step. Define the scalar sum (corresponding to the logits, if we are using binary classification) as $\eta_t = \mathbf{w}_t^\top \mathbf{x}_t$. If $p_{t|t-1}(\mathbf{w}_t) = \prod_j \mathcal{N}(w_t^j | \mu_{t|t-1}^j, \tau_{t|t-1}^j)$, then we can compute the 1d prior predictive distribution for η_t as follows:

$$p(\eta_t | \mathcal{D}_{1:t-1}, \mathbf{x}_t) \approx p_{t|t-1}(\eta_t) = \mathcal{N}(\eta_t | m_{t|t-1}, v_{t|t-1}) \quad (8.212)$$

$$m_{t|t-1} = \sum_j x_{t,j} \mu_{t|t-1}^j \quad (8.213)$$

$$v_{t|t-1} = \sum_j x_{t,j}^2 \tau_{t|t-1}^j \quad (8.214)$$

The posterior for the 1d η_t is given by

$$p(\eta_t | \mathcal{D}_{1:t}) \approx p_t(\eta_t) = \mathcal{N}(\eta_t | m_t, v_t) \quad (8.215)$$

$$m_t = \int \eta_t \frac{1}{Z_t} p(y_t | \eta_t) p_{t|t-1}(\eta_t) d\eta_t \quad (8.216)$$

$$v_t = \int \eta_t^2 \frac{1}{Z_t} p(y_t | \eta_t) p_{t|t-1}(\eta_t) d\eta_t - m_t^2 \quad (8.217)$$

$$Z_t = \int p(y_t | \eta_t) p_{t|t-1}(\eta_t) d\eta_t \quad (8.218)$$

where $p(y_t | \eta_t) = \text{Ber}(y_t | \eta_t)$. These integrals are one dimensional, and so can be efficiently computed using Gaussian quadrature, as explained in [Zoe07; KB00].

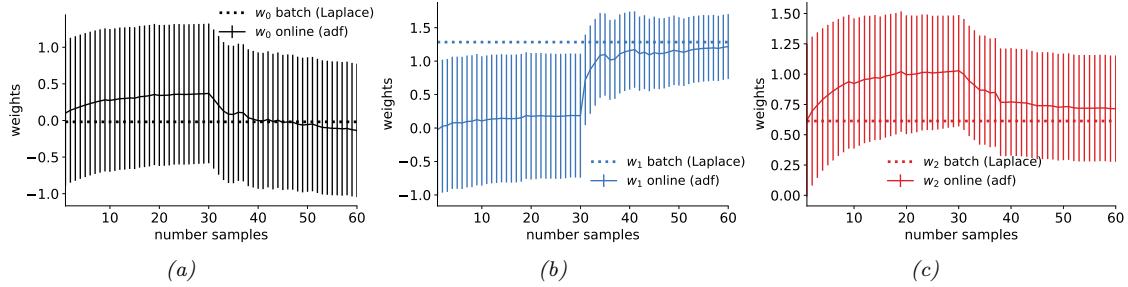


Figure 8.10: Bayesian inference applied to a 2d binary logistic regression problem, $p(y = 1|\mathbf{x}) = \sigma(w_0 + w_1x_1 + w_2x_2)$. We show the marginal posterior mean and variance for each parameter vs time as computed by ADF. The dotted horizontal line is the offline Laplace approximation. Generated by `adf_logistic_regression_demo.ipynb`.

Having inferred $p_t(\eta_t)$, we need to compute $p_t(\mathbf{w}|\eta_t)$. This can be done as follows. Define δ_m as the change in the mean and δ_v as the change in the variance:

$$m_t = m_{t|t-1} + \delta_m, \quad v_t = v_{t|t-1} + \delta_v \quad (8.219)$$

Using the fact that $p(\eta_t|\mathbf{w}) = \mathcal{N}(\eta_t|\mathbf{w}^\top \eta_t, 0)$ is a linear Gaussian system, with prior $p(\mathbf{w}) = p(\mathbf{w}|\boldsymbol{\mu}_{t|t-1}, \boldsymbol{\tau}_{t|t-1})$ and “soft evidence” $p(\eta_t) = \mathcal{N}(m_t, v_t)$, we can derive the posterior for $p(\mathbf{w}|\mathcal{D}_t)$ as follows:

$$p_t(w_t^i) = \mathcal{N}(w_t^i | \mu_{t|t}^i, \tau_{t|t}^i) \quad (8.220)$$

$$\mu_{t|t}^i = \mu_{t|t-1}^i + a_i \delta_m \quad (8.221)$$

$$\tau_{t|t}^i = \tau_{t|t-1}^i + a_i^2 \delta_v \quad (8.222)$$

$$a_i \triangleq \frac{x_t^i \tau_{t|t-1}^i}{\sum_j (x_t^j)^2 + \tau_{t|t-1}^j} \quad (8.223)$$

Thus we see that the parameters which correspond to inputs i with larger magnitude (big $|x_t^i|$) or larger uncertainty (big $\tau_{t|t-1}^i$) get updated most, due to a large a_i factor, which makes intuitive sense.

As an example, we consider a 2d binary classification problem. We sequentially compute the posterior using the ADF, and compare to the offline estimate computed using a Laplace approximation. In Figure 8.10 we plot the posterior marginals over the 3 parameters as a function of “time” (i.e., after conditioning on each training example one). We see that we converge to the offline MAP estimate. In Figure 8.11, we show the results of performing sequential Bayesian updating in a different ordering of the data. We still converge to approximate the same answer. In Figure 8.12, we see that the resulting posterior predictive distributions from the Laplace estimate and ADF estimate (at the end of training) are similar.

Note that the whole algorithm only takes $O(D)$ time and space per step, the same as SGD. However, unlike SGD, there are no step-size parameters, since the diagonal covariance implicitly specifies the size of the update for each dimension. Furthermore, we get a posterior approximation, not just a point estimate.

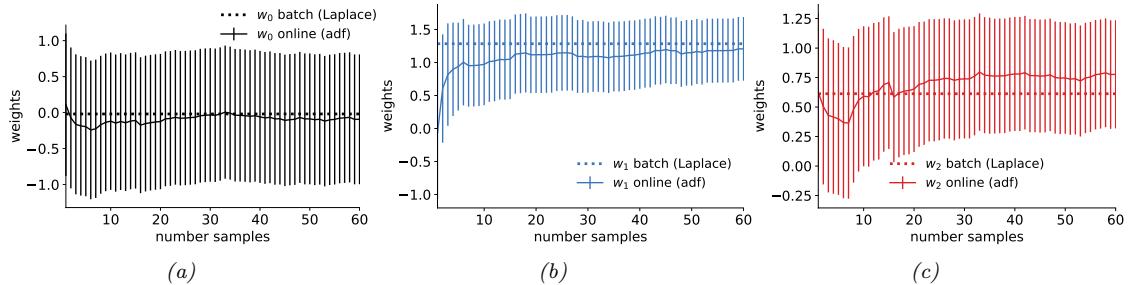


Figure 8.11: Same as Figure 8.10, except the order in which the data is visited is different. Generated by `adf_logistic_regression_demo.ipynb`.

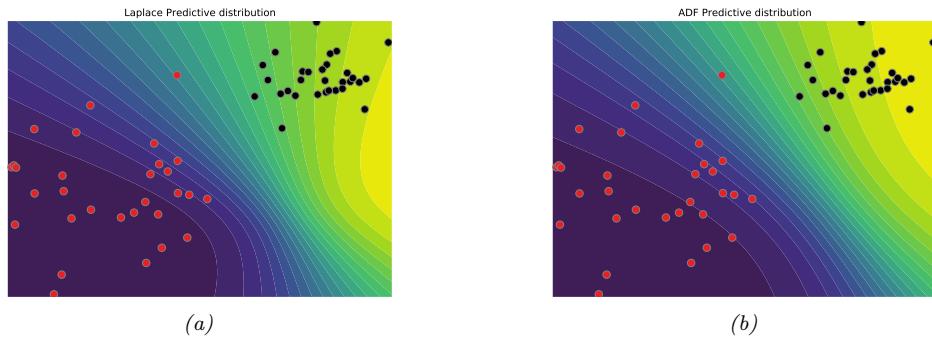


Figure 8.12: Predictive distribution for the binary logistic regression problem. (a) Result from Laplace approximation. (b) Result from ADF at the final step. Generated by `adf_logistic_regression_demo.ipynb`.

The overall approach is very similar to the generalized posterior linearization filter of Section 8.5.3, which uses quadrature (or the unscented transform) to compute a Gaussian approximation to the joint $p(y_t, \mathbf{w}_t | \mathcal{D}_{1:t-1})$, from which we can easily compute $p(\mathbf{w}_t | \mathcal{D}_{1:t})$. However, ADF approximates the posterior rather than the joint, as explained in Section 8.6.1.

8.6.4 ADF for online DNNs

In Section 17.5.3, we show how to use ADF to recursively approximate the posterior over the parameters of a deep neural network in an online fashion. This generalizes Section 8.6.3 to the case of nonlinear models.

8.7 Other inference methods for SSMs

There are a variety of other inference algorithms that can be applied to SSMs. We give a very brief summary below. For more details, see e.g., [Dau05; Sim06; Fra08; Sar13; SS23; Tri21].

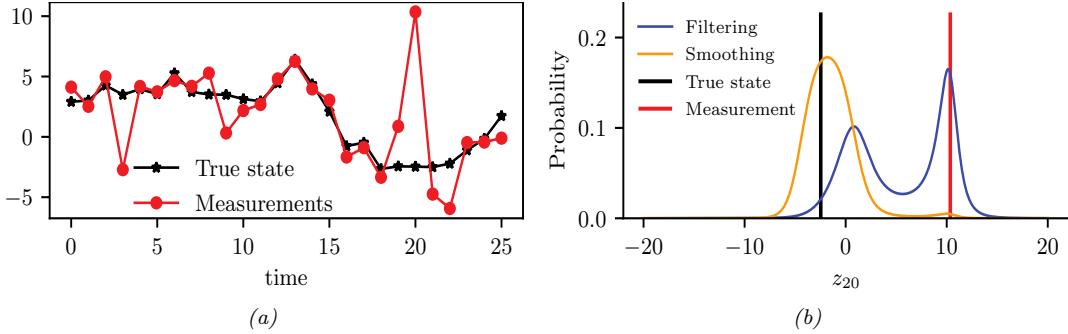


Figure 8.13: (a) Observations and true and estimated state. (b) Marginal distributions for time step $t = 20$. Generated by [discretized_ssm_student.ipynb](#).

8.7.1 Grid-based approximations

A very simple approach to approximate inference in SSMs is to discretize the state space, and then to apply the HMM filter and smoother (see Section 9.2.3), as proposed in [RG17]. This is called a **grid-based approximation**. Unfortunately, this approach will not scale to higher dimensional problems, due to the curse of dimensionality. In particular, we know that the HMM filter takes $O(K^2)$ operations per time step, if there are K states. If we have N_z dimensions, each discretized into B bins, then we have $K = B^{N_z}$, so the approach quickly becomes intractable.

However, this approach can be useful in 1d or 2d. As an illustration, consider a simple 1d SSM with linear dynamics corrupted by additive Student noise:

$$z_t = z_{t-1} + \mathcal{T}_2(0, 1) \quad (8.224)$$

The observations are also linear, and are also corrupted by additive Student noise:

$$y_t = z_t + \mathcal{T}_2(0, 1) \quad (8.225)$$

This robust observation model is useful when there are potential outliers in the observed data, such as at time $t = 20$ in Figure 8.13a. (See also Section 8.5.5 for discussion of robust Kalman filters.)

Unfortunately the use of a non-Gaussian likelihood means that the resulting posterior can become multimodal. Fortunately, this is not a problem for the grid-based approach. We show the results for filtering and smoothing in Figure 8.14a and in Figure 8.14b. We see that at $t = 20$, the filtering distribution, $p(z_t | \mathbf{y}_{1:20})$, is bimodal, with a mean that is quite far from the true state (see Figure 8.13b for a detailed plot). Such a multimodal distribution can be approximated by a suitably fine discretization.

8.7.2 Expectation propagation

In Section 10.7 we discuss the expectation propagation (EP) algorithm, which can be viewed as an iterative version of ADF (Section 8.6). In particular, at each step we combine each exact local likelihood factor with approximate factors from both the past filtering distribution and the future smoothed posterior; these factors are combined to compute the locally exact posterior, which is then

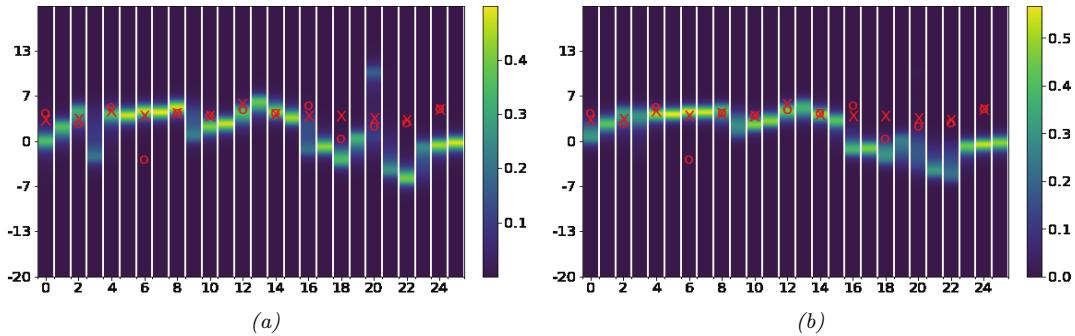


Figure 8.14: Discretized posterior of the latent state at each time step. Red cross is the true latent state. Red circle is observation. (a) Filtering. (b) Smoothing. Generated by [discretized_ssm_student.ipynb](#).

projected back to the tractable family (e.g., Gaussian), before moving to the next time step. This process can be iterated for increased accuracy. In many cases the local EP update is intractable, but we can make a local Gaussian approximation, similar to the one in general Gaussian filtering (Section 8.5.1), as explained in [Kam+22].

8.7.3 Variational inference

EP can be viewed as locally minimizing the inclusive KL, $D_{\text{KL}}(p(\mathbf{z}_t|\mathbf{y}_{1:T}) \parallel q(\mathbf{z}_t|\mathbf{y}_{1:T}))$, for each time step t . An alternative approach is to globally minimize the exclusive KL, $D_{\text{KL}}(q(\mathbf{z}_{1:T}|\mathbf{y}_{1:T}) \parallel p(\mathbf{z}_{1:T}|\mathbf{y}_{1:T}))$; this is called variational inference, and is explained in Chapter 10. The difference between these two objectives is discussed in more detail in Section 5.1.4.1, but from a practical point of view, the main advantage of VI is that we can derive a tractable lower bound to the objective, and can then optimize it using stochastic optimization. This method is guaranteed to converge, unlike EP. For more details on VI applied to SSMs (both state estimation and parameter estimation), see e.g., [CWS21; Cou+20; Cou+21; BFY20; FLMM21; Cam+21].

8.7.4 MCMC

In Chapter 12 we discuss Markov chain Monte Carlo (MCMC) methods, which can be used to draw samples from intractable posteriors. In the case of SSMs, this includes both the distribution over states, $p(\mathbf{z}_{1:T}|\mathbf{y}_{1:T})$, and the distribution over parameters, $p(\boldsymbol{\theta}|\mathbf{y}_{1:T})$. In some cases, such as when using HMMs or linear-Gaussian SSMs, we can perform blocked Gibbs sampling, in which we use forwards filtering backwards sampling to sample an entire sequence from $p(\mathbf{z}_{1:T}|\mathbf{y}_{1:T}, \boldsymbol{\theta})$, followed by sampling the parameters, $p(\boldsymbol{\theta}|\mathbf{z}_{1:T}, \mathbf{y}_{1:T})$ (see e.g., [CK96; Sco02; CMR05] for details.) Alternatively we can marginalize out the hidden states and just compute the parameter posterior $p(\boldsymbol{\theta}|\mathbf{y}_{1:T})$. When state inference is intractable, we can use gradient-based HMC methods (assuming the states are continuous), although this does not scale well to long sequences.

8.7.5 Particle filtering

In Section 13.2 we discuss particle filtering, which is a form of sequential Bayesian inference for SSMs which replaces the assumption that the posterior is (approximately) Gaussian with a more flexible representation, namely a set of weighted samples called “particles” (see e.g., [Aru+02; DJ11; NLS19]). Essentially the technique amounts to a form of importance sampling, combined with steps to prevent “particle impoverishment”, which refers to some samples receiving negligible weight because they are too improbable in the posterior (which grows with time). Particle filtering is widely used because it is very flexible, and has good theoretical properties. In practice it may require many samples to get a good approximation, but we can use heuristic methods, such as the extended or unscented Kalman filters, as proposal distributions, which can improve the efficiency significantly. In the offline setting, we can use particle smoothing (Section 13.5) or SMC (sequential Monte Carlo) samplers (Section 13.6).

9 Message passing algorithms

9.1 Introduction

In this chapter we consider posterior inference (i.e., computing marginals, modes, samples, etc) for probability distributions that can be represented by a probabilistic graphical model (PGM, Chapter 4) with some kind of sparse graph structure (i.e., it is not a fully connected graph). The algorithms we discuss will leverage the conditional independence properties encoded in the graph structure (discussed in Chapter 4) in order to perform efficient inference. In particular, we will use the principle of **dynamic programming** (DP), which finds an optimal solution by solving subproblems and then combining them.

DP can be implemented by computing local quantities for each node (or clique) in the graph, and then sending **messages** to neighboring nodes (or cliques) so that all nodes (cliques) can come to an overall consensus about the global solutions. Hence these are known as **message passing algorithms**. Each message can be interpreted as probability distribution about the value of a node given evidence from part of the graph. These distributions are often called **belief states**, so these algorithms are also called **belief propagation (BP)** algorithms.

In Section 9.2, we consider the special case where the graph structure is a 1d chain, which is an important special case. (For a chain, a natural approach is to send messages forwards in time, and then backwards in time, so this method can also be used for inference in state space models, as we discuss in Chapter 8.) In Section 9.3, we can generalize this approach to work with trees, and in Section 9.4, we generalize it to work with any graph, including ones with cycles or loops. However, sending messages on loopy graphs may give incorrect answers. In such cases, we may wish to convert the graph to a tree, and then send messages on it, using the methods discussed in Section 9.5 and Section 9.6. We can also pose the inference problem as an optimization problem, as we discuss in Section 9.7.

9.2 Belief propagation on chains

In this section, we consider inference for PGMs where the graph structure is a 1d chain. For notational simplicity, we focus on the case where the graphical model is directed rather than undirected, although the resulting methods are easy to generalize. In addition, we only consider the case where all the hidden variables are discrete; we discuss generalizations to handle continuous latent variables in Chapter 8 and Chapter 13.

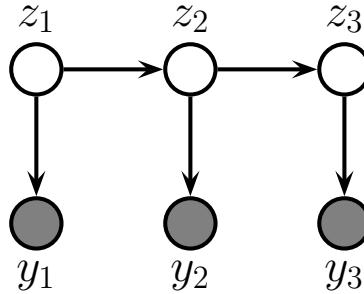


Figure 9.1: An HMM represented as a graphical model. z_t are the hidden variables at time t , y_t are the observations (outputs).

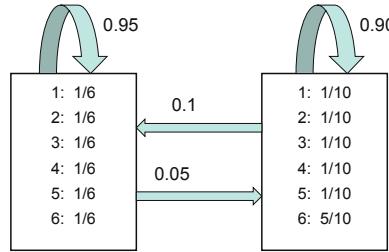


Figure 9.2: The state transition matrix \mathbf{A} and observation matrix \mathbf{B} for the casino HMM. Adapted from [Dur+98, p54].

9.2.1 Hidden Markov Models

In this section we assume the graphical model can be represented as a state space model, as shown in Figure 9.1. We discuss SSMs in more detail in Chapter 29, but we can think of them as latent variable sequence models with the conditional independencies shown by the chain-structured graphical model Figure 8.1. The corresponding joint distribution has the form

$$p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T}) = \left[p(\mathbf{z}_1) \prod_{t=2}^T p(\mathbf{z}_t | \mathbf{z}_{t-1}) \right] \left[\prod_{t=1}^T p(\mathbf{y}_t | \mathbf{z}_t) \right] \quad (9.1)$$

where \mathbf{z}_t are the hidden variables at time t , and \mathbf{y}_t are the observations (outputs). If all the latent variables are discrete (as we assume in this section), the resulting model is called a **hidden Markov model** or **HMM**. We consider SSMs with continuous latent variables in Chapter 8.

9.2.1.1 Example: casino HMM

As a concrete example from [Dur+98], we consider the **occasionally dishonest casino**. We assume we are in a casino and observe a series of die rolls, $y_t \in \{1, 2, \dots, 6\}$. Being a keen-eyed statistician, we notice that the distribution of values is not what we expect from a fair die: it seems that there

are occasional “streaks”, in which 6s seem to show up more often than other values. We would like to estimate the underlying state, namely whether the die is fair or loaded, so that we make predictions about the future.

To formalize this, let $z_t \in \{1, 2\}$ represent the unknown hidden state (fair or loaded) at time t , and let $y_t \in \{1, \dots, 6\}$ represent the observed outcome (die roll). Let $A_{jk} = p(z_t = k | z_{t-1} = j)$ be the state transition matrix. Most of the time the casino uses a fair die, $z = 1$, but occasionally it switches to a loaded die, $z = 2$, for a short period, as shown in the state transition diagram in Figure 9.2.

Let $B_{kl} = p(y_t = l | z_t = k)$ be the observation matrix corresponding to a categorical distribution over values of the die face. If $z = 1$ the observation distribution is a uniform categorical distribution over the symbols $\{1, \dots, 6\}$. If $z = 2$, the observation distribution is skewed towards face 6. That is,

$$p(y_t | z_t = 1) = \text{Cat}(y_t | [1/6, \dots, 1/6]) \quad (9.2)$$

$$p(y_t|z_t = 2) = \text{Cat}(y_t|[1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 5/10]) \quad (9.3)$$

If we sample from this model, we may generate data such as the following:

Here `obs` refers to the observation and `hid` refers to the hidden state (1 is fair and 2 is loaded). In the full sequence of length 300, we find the empirical fraction of times that we observe a 6 in hidden state 1 to be 0.149, and in state 2 to be 0.472, which are very close to the expected fractions. (See `casino_hmm.ipynb` for the code.)

9.2.1.2 Posterior inference

Our goal is to infer the hidden states by computing the posterior over all the hidden nodes in the model, $p(z_t|y_{1:T})$. This is called the **smoothing distribution**. By the Markov property, we can break this into two terms:

$$p(\mathbf{z}_t = j | \mathbf{y}_{t+1:T}, \mathbf{y}_{1:t}) \propto p(\mathbf{z}_t = j, \mathbf{y}_{t+1:T} | \mathbf{y}_{1:t}) = p(\mathbf{z}_t = j | \mathbf{y}_{1:t}) p(\mathbf{y}_{t+1:T} | \mathbf{z}_t = j, \mathbf{y}_{1:t}) \quad (9.4)$$

We will first compute the **filtering distribution** $p(\mathbf{z}_t = j | \mathbf{y}_{1:t})$ by working forwards in time. We then compute the $p(\mathbf{y}_{t+1:T} | \mathbf{z}_t = j)$ terms by working backwards in time, and then we finally combine both terms. Both passes take (TK^2) time, where K is the number of discrete hidden states. We give the details below.

9.2.2 The forwards algorithm

As we discuss in Section 8.1.2, the **Bayes filter** is an algorithm for recursively computing the **belief state** $p(z_t|y_{1:t})$ given the prior belief from the previous step, $p(z_{t-1}|y_{1:t-1})$, the new observation y_t , and the model. In the HMM literature, this is known as the **forwards algorithm**.

In an HMM, the latent states z_t are discrete, so we can define the belief state as a vector, $\alpha_t(j) \triangleq p(z_t = j | \mathbf{y}_{1:t})$, the local evidence as another vector, $\lambda_t(j) \triangleq p(\mathbf{y}_t | z_t = j)$, and the transition matrix as $A_{i,j} = p(z_t = j | z_{t-1} = i)$. Then the predict step becomes

$$\alpha_{t|t-1}(j) \triangleq p(z_t = j | \mathbf{y}_{1:t-1}) = \sum_i p(z_t = j | z_{t-1} = i) p(z_{t-1} = i | \mathbf{y}_{1:t-1}) = \sum_i A_{i,j} \alpha_{t-1}(i) \quad (9.5)$$

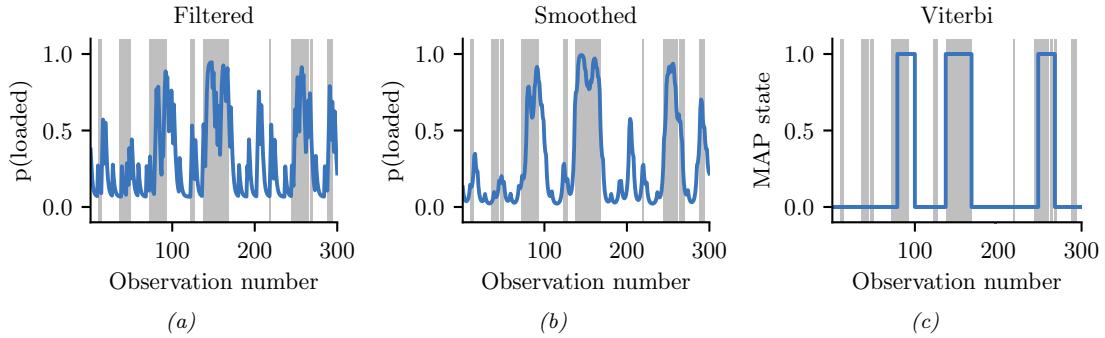


Figure 9.3: Inference in the dishonest casino. Vertical gray bars denote times when the hidden state corresponded to the loaded die. Blue lines represent the posterior probability of being in that state given different subsets of observed data. If we recover the true state exactly, the blue curve will transition at the same time as the gray bars. (a) Filtered estimates. (b) Smoothed estimates. (c) MAP trajectory. Generated by [casino_hmm.ipynb](#).

and the update step becomes

$$\alpha_t(j) = \frac{1}{Z_t} p(\mathbf{y}_t | z_t = j) p(z_t = j | \mathbf{y}_{1:t-1}) = \frac{1}{Z_t} \lambda_t(j) \alpha_{t|t-1}(j) = \frac{1}{Z_t} \lambda_t(j) \left[\sum_i \alpha_{t-1}(i) A_{i,j} \right] \quad (9.6)$$

where the normalization constant for each time step is given by

$$Z_t \triangleq p(\mathbf{y}_t | \mathbf{y}_{1:t-1}) = \sum_{j=1}^K p(\mathbf{y}_t | z_t = j) p(z_t = j | \mathbf{y}_{1:t-1}) = \sum_{j=1}^K \lambda_t(j) \alpha_{t|t-1}(j) \quad (9.7)$$

We can write the update equation in matrix-vector notation as follows:

$$\boldsymbol{\alpha}_t = \text{normalize}(\boldsymbol{\lambda}_t \odot (\mathbf{A}^\top \boldsymbol{\alpha}_{t-1})) \quad (9.8)$$

where \odot represents elementwise vector multiplication, and the normalize function just ensures its argument sums to one. (See Section 9.2.3.4 for more discussion on normalization.)

Figure 9.3(a) illustrates filtering for the casino HMM, applied to a random sequence $\mathbf{y}_{1:T}$ of length $T = 300$. In blue, we plot the probability that the die is in the loaded (vs fair) state, based on the evidence seen so far. The gray bars indicate time intervals during which the generative process actually switched to the loaded die. We see that the probability generally increases in the right places.

9.2.3 The forwards-backwards algorithm

In this section, we present the most common approach to smoothing in HMMs, known as the **forwards-backwards** or **FB** algorithm [Rab89]. In the forwards pass, we compute $\alpha_t(j) = p(z_t = j | \mathbf{y}_{1:t})$ as before. In the backwards pass, we compute the conditional likelihood

$$\beta_t(j) \triangleq p(\mathbf{y}_{t+1:T} | z_t = j) \quad (9.9)$$

We then combine these using

$$\gamma_t(j) = p(\mathbf{z}_t = j | \mathbf{y}_{t+1:T}, \mathbf{y}_{1:t}) \propto p(\mathbf{z}_t = j, \mathbf{y}_{t+1:T} | \mathbf{y}_{1:t}) \quad (9.10)$$

$$= p(\mathbf{z}_t = j | \mathbf{y}_{1:t}) p(\mathbf{y}_{t+1:T} | \mathbf{z}_t = j, \mathbf{y}_{1:t}) = \alpha_t(j) \beta_t(j) \quad (9.11)$$

In matrix notation, this becomes

$$\boldsymbol{\gamma}_t = \text{normalize}(\boldsymbol{\alpha}_t \odot \boldsymbol{\beta}_t) \quad (9.12)$$

Note that the forwards and backwards passes can be computed independently, but both need access to the local evidence $p(\mathbf{y}_t | \mathbf{z}_t)$. The results are only combined at the end. This is therefore called **two-filter smoothing** [Kit04].

9.2.3.1 Backwards recursion

We can recursively compute the β 's in a right-to-left fashion as follows:

$$\beta_{t-1}(i) = p(\mathbf{y}_{t:T} | \mathbf{z}_{t-1} = i) \quad (9.13)$$

$$= \sum_j p(\mathbf{z}_t = j, \mathbf{y}_t, \mathbf{y}_{t+1:T} | \mathbf{z}_{t-1} = i) \quad (9.14)$$

$$= \sum_j p(\mathbf{y}_{t+1:T} | \mathbf{z}_t = j, \mathbf{y}_t, \mathbf{z}_{t-1} = i) p(\mathbf{z}_t = j, \mathbf{y}_t | \mathbf{z}_{t-1} = i) \quad (9.15)$$

$$= \sum_j p(\mathbf{y}_{t+1:T} | \mathbf{z}_t = j) p(\mathbf{y}_t | \mathbf{z}_t = j, \mathbf{z}_{t-1} = i) p(\mathbf{z}_t = j | \mathbf{z}_{t-1} = i) \quad (9.16)$$

$$= \sum_j \beta_t(j) \lambda_t(j) A_{i,j} \quad (9.17)$$

We can write the resulting equation in matrix-vector form as

$$\boldsymbol{\beta}_{t-1} = \mathbf{A}(\boldsymbol{\lambda}_t \odot \boldsymbol{\beta}_t) \quad (9.18)$$

The base case is

$$\beta_T(i) = p(\mathbf{y}_{T+1:T} | \mathbf{z}_T = i) = p(\emptyset | \mathbf{z}_T = i) = 1 \quad (9.19)$$

which is the probability of a non-event.

Note that $\boldsymbol{\beta}_t$ is not a probability distribution over states, since it does not need to satisfy $\sum_j \beta_t(j) = 1$. However, we usually normalize it to avoid numerical underflow (see Section 9.2.3.4).

9.2.3.2 Example

In Figure 9.3(a-b), we compare filtering and smoothing for the casino HMM. We see that the posterior distributions when conditioned on all the data (past and future) are indeed smoother than when just conditioned on the past (filtering).

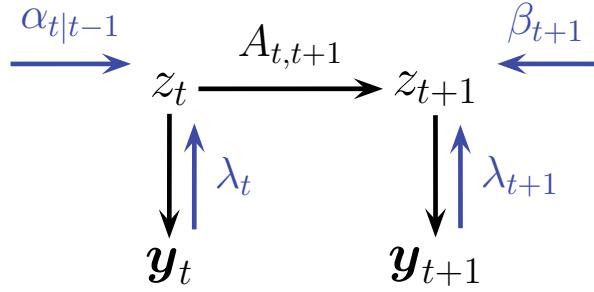


Figure 9.4: Computing the two-slice joint distribution for an HMM from the forwards messages, backwards messages, and local evidence messages.

9.2.3.3 Two-slice smoothed marginals

We can compute the two-slice marginals using the output of the forwards-backwards algorithm as follows:

$$p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:T}) = p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}, \mathbf{y}_{t+1:T}) \quad (9.20)$$

$$\propto p(\mathbf{y}_{t+1:T} | \mathbf{z}_t, \mathbf{z}_{t+1}, \mathbf{y}_{1:t}) p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}) \quad (9.21)$$

$$= p(\mathbf{y}_{t+1:T} | \mathbf{z}_{t+1}) p(\mathbf{z}_t, \mathbf{z}_{t+1} | \mathbf{y}_{1:t}) \quad (9.22)$$

$$= p(\mathbf{y}_{t+1:T} | \mathbf{z}_{t+1}) p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{z}_t) \quad (9.23)$$

$$= p(\mathbf{y}_{t+1}, \mathbf{y}_{t+2:T} | \mathbf{z}_{t+1}) p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{z}_t) \quad (9.24)$$

$$= p(\mathbf{y}_{t+1} | \mathbf{z}_{t+1}) p(\mathbf{y}_{t+2:T} | \mathbf{z}_{t+1}, \mathbf{y}_{t+1}) p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{z}_t) \quad (9.25)$$

$$= p(\mathbf{y}_{t+1} | \mathbf{z}_{t+1}) p(\mathbf{y}_{t+2:T} | \mathbf{z}_{t+1}) p(\mathbf{z}_t | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} | \mathbf{z}_t) \quad (9.26)$$

We can rewrite this in terms of the already computed quantities as follows:

$$\xi_{t,t+1}(i, j) \propto \lambda_{t+1}(j) \beta_{t+1}(j) \alpha_t(i) A_{i,j} \quad (9.27)$$

Or in matrix-vector form:

$$\boldsymbol{\xi}_{t,t+1} \propto \mathbf{A} \odot [\boldsymbol{\alpha}_t (\boldsymbol{\lambda}_{t+1} \odot \boldsymbol{\beta}_{t+1})^\top] \quad (9.28)$$

Since $\boldsymbol{\alpha}_t \propto \boldsymbol{\lambda}_t \odot \boldsymbol{\alpha}_{t|t-1}$, we can also write the above equation as follows:

$$\boldsymbol{\xi}_{t,t+1} \propto \mathbf{A} \odot [(\boldsymbol{\lambda}_t \odot \boldsymbol{\alpha}_{t|t-1}) \odot (\boldsymbol{\lambda}_{t+1} \odot \boldsymbol{\beta}_{t+1})^\top] \quad (9.29)$$

This can be interpreted as a product of incoming messages and local factors, as shown in Figure 9.4. In particular, we combine the factors $\boldsymbol{\alpha}_{t|t-1} = p(\mathbf{z}_t | \mathbf{y}_{1:t-1})$, $\mathbf{A} = p(\mathbf{z}_{t+1} | \mathbf{z}_t)$, $\boldsymbol{\lambda}_t \propto p(\mathbf{y}_t | \mathbf{z}_t)$, $\boldsymbol{\lambda}_{t+1} \propto p(\mathbf{y}_{t+1} | \mathbf{z}_{t+1})$, and $\boldsymbol{\beta}_{t+1} \propto p(\mathbf{y}_{t+2:T} | \mathbf{z}_{t+1})$ to get $p(\mathbf{z}_t, \mathbf{z}_{t+1}, \mathbf{y}_t, \mathbf{y}_{t+1}, \mathbf{y}_{t+2:T} | \mathbf{y}_{1:t-1})$, which we can then normalize.

9.2.3.4 Numerically stable implementation

In most publications on HMMs, such as [Rab89], the forwards message is defined as the following unnormalized *joint* probability:

$$\alpha'_t(j) = p(\mathbf{z}_t = j, \mathbf{y}_{1:t}) = \lambda_t(j) \left[\sum_i \alpha'_{t-1}(i) A_{i,j} \right] \quad (9.30)$$

We instead define the forwards message as the normalized *conditional* probability

$$\alpha_t(j) = p(\mathbf{z}_t = j | \mathbf{y}_{1:t}) = \frac{1}{Z_t} \lambda_t(j) \left[\sum_i \alpha_{t-1}(i) A_{i,j} \right] \quad (9.31)$$

The unnormalized (joint) form has several problems. First, it rapidly suffers from numerical underflow, since the probability of the joint event that $(\mathbf{z}_t = j, \mathbf{y}_{1:t})$ is vanishingly small.¹ Second, it is less interpretable, since it is not a distribution over states. Third, it precludes the use of approximate inference methods that try to approximate posterior distributions (we will see such methods later). We therefore always use the normalized (conditional) form.

Of course, the two definitions only differ by a multiplicative constant, since $p(\mathbf{z}_t = j | \mathbf{y}_{1:t}) = p(\mathbf{z}_t = j, \mathbf{y}_{1:t}) / p(\mathbf{y}_{1:t})$ [Dev85]. So the *algorithmic* difference is just one line of code (namely the presence or absence of a call to the `normalize` function). Nevertheless, we feel it is better to present the normalized version, since it will encourage readers to implement the method properly (i.e., normalizing after each step to avoid underflow).

In practice it is more numerically stable to compute the log probabilities $\ell_t(j) = \log p(\mathbf{y}_t | \mathbf{z}_t = j)$ of the evidence, rather than the probabilities $\lambda_t(j) = p(\mathbf{y}_t | \mathbf{z}_t = j)$. We can combine the state conditional log likelihoods $\lambda_t(j)$ with the state prior $p(\mathbf{z}_t = j | \mathbf{y}_{1:t-1})$ by using the log-sum-exp trick, as in Equation (28.30).

9.2.4 Forwards filtering backwards smoothing

An alternative way to perform offline smoothing is to use forwards filtering/backwards smoothing, as discussed in Section 8.1.3. In this approach, we first perform the forwards or filtering pass, and then compute the smoothed belief states by working backwards, from right (time $t = T$) to left ($t = 1$). This approach is widely used for SSMs with continuous latent states, since the backwards likelihood $\beta_t(i)$ used in Section 9.2.3 is not always well defined when the state space is not discrete.

We assume by induction that we have already computed

$$\gamma_{t+1}(j) \triangleq p(\mathbf{z}_{t+1} = j | \mathbf{y}_{1:T}) \quad (9.32)$$

1. For example, if the observations are independent of the states, we have $p(\mathbf{z}_t = j, \mathbf{y}_{1:t}) = p(\mathbf{z}_t = j) \prod_{i=1}^t p(\mathbf{y}_i)$, which becomes exponentially small with t .

We then compute the smoothed joint distribution over two consecutive time steps:

$$\xi_{t,t+1}(i, j) \triangleq p(\mathbf{z}_t = i, \mathbf{z}_{t+1} = j | \mathbf{y}_{1:T}) = p(\mathbf{z}_t = i | \mathbf{z}_{t+1} = j, \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} = j | \mathbf{y}_{1:T}) \quad (9.33)$$

$$= \frac{p(\mathbf{z}_{t+1} = j | \mathbf{z}_t = i) p(\mathbf{z}_t = i | \mathbf{y}_{1:t}) p(\mathbf{z}_{t+1} = j | \mathbf{y}_{1:T})}{p(\mathbf{z}_{t+1} = j | \mathbf{y}_{1:t})} \quad (9.34)$$

$$= \alpha_t(i) A_{i,j} \frac{\gamma_{t+1}(j)}{\alpha_{t+1|t}(j)} \quad (9.35)$$

where

$$\alpha_{t+1|t}(j) = p(\mathbf{z}_{t+1} = j | \mathbf{y}_{1:t}) = \sum_{i'} A(i', j) \alpha_t(i') \quad (9.36)$$

is the one-step-ahead predictive distribution. We can interpret the ratio in Equation (9.35) as dividing out the old estimate of \mathbf{z}_{t+1} given $\mathbf{y}_{1:t}$, namely $\alpha_{t+1|t}$, and multiplying in the new estimate given $\mathbf{y}_{1:T}$, namely γ_{t+1} .

Once we have the two sliced smoothed distribution, we can easily get the marginal one slice smoothed distribution using

$$\gamma_t(i) = p(\mathbf{z}_t = i | \mathbf{y}_{1:T}) = \sum_j \xi_{t,t+1}(i, j) = \alpha_t(i) \sum_j \left[A_{i,j} \frac{\gamma_{t+1}(j)}{\alpha_{t+1|t}(j)} \right] \quad (9.37)$$

We initialize the recursion using $\gamma_T(j) = \alpha_T(j) = p(\mathbf{z}_T = j | \mathbf{y}_{1:T})$.

9.2.5 Time and space complexity

It is clear that a straightforward implementation of the forwards-backwards algorithm takes $O(K^2 T)$ time, since we must perform a $K \times K$ matrix multiplication at each step. For some applications, such as speech recognition, K is very large, so the $O(K^2)$ term becomes prohibitive. Fortunately, if the transition matrix is sparse, we can reduce this substantially. For example, in a sparse left-to-right transition matrix (e.g., Figure 9.6(a)), the algorithm takes $O(TK)$ time.

In some cases, we can exploit special properties of the state space, even if the transition matrix is not sparse. In particular, suppose the states represent a discretization of an underlying continuous state-space, and the transition matrix has the form $A_{i,j} \propto \rho(\mathbf{z}_j - \mathbf{z}_i)$, where \mathbf{z}_i is the continuous vector represented by state i and $\rho(u)$ is some scalar cost function, such as Euclidean distance. Then one can implement the forwards-backwards algorithm in $O(TK \log K)$ time. The key is to rewrite Equation (9.5) as a convolution,

$$\alpha_{t|t-1}(j) = p(\mathbf{z}_t = j | \mathbf{y}_{1:t-1}) = \sum_i \alpha_{t-1}(i) A_{i,j} = \sum_i \alpha_{t-1}(i) \rho(j - i) \quad (9.38)$$

and then to apply the Fast Fourier Transform. (A similar transformation can be applied in the backwards pass.) This is very useful for models with large state spaces. See [FHK03] for details.

We can also reduce inference to $O(\log T)$ time by using a **parallel prefix scan** operator that can be run efficiently on GPUs. For details, see [HSGF21].

In some cases, the bottleneck is memory, not time. In particular, to compute the posteriors γ_t , we must store the filtered distributions α_t for $t = 1, \dots, T$ until we do the backwards pass. It is possible

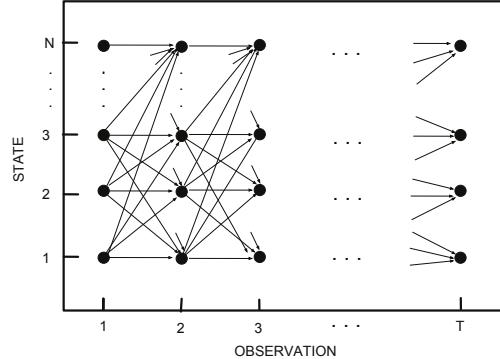


Figure 9.5: The trellis of states vs time for a Markov chain. Adapted from [Rab89].

to devise a simple divide-and-conquer algorithm that reduces the space complexity from $O(KT)$ to $O(K \log T)$ at the cost of increasing the running time from $O(K^2T)$ to $O(K^2T \log T)$. The basic idea is to store α_t and β_t vectors at a logarithmic number of intermediate checkpoints, and then recompute the missing messages on demand from these checkpoints. See [BMR97; ZP00] for details.

9.2.6 The Viterbi algorithm

The MAP estimate is (one of) the sequences with maximum posterior probability:

$$\mathbf{z}_{1:T}^* = \underset{\mathbf{z}_{1:T}}{\operatorname{argmax}} p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) = \underset{\mathbf{z}_{1:T}}{\operatorname{argmax}} \log p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) \quad (9.39)$$

$$= \underset{\mathbf{z}_{1:T}}{\operatorname{argmax}} \log \pi_1(\mathbf{z}_1) + \log \lambda_1(\mathbf{z}_1) + \sum_{t=2}^T [\log A(\mathbf{z}_{t-1}, \mathbf{z}_t) + \log \lambda_t(\mathbf{z}_t)] \quad (9.40)$$

This is equivalent to computing a shortest path through the **trellis diagram** in Figure 9.5, where the nodes are possible states at each time step, and the node and edge weights are log probabilities. This can be computed in $O(TK^2)$ time using the **Viterbi algorithm** [Vit67], as we explain below.

9.2.6.1 Forwards pass

Recall the (unnormalized) forwards equation

$$\alpha'_t(j) = p(\mathbf{z}_t = j, \mathbf{y}_{1:t}) = \sum_{\mathbf{z}_1, \dots, \mathbf{z}_{t-1}} p(\mathbf{z}_{1:t-1}, \mathbf{z}_t = j, \mathbf{y}_{1:t}) \quad (9.41)$$

Now suppose we replace sum with max to get

$$\delta_t(j) \triangleq \max_{\mathbf{z}_1, \dots, \mathbf{z}_{t-1}} p(\mathbf{z}_{1:t-1}, \mathbf{z}_t = j, \mathbf{y}_{1:t}) \quad (9.42)$$

This is the maximum probability we can assign to the data so far if we end up in state j . The key insight is that the most probable path to state j at time t must consist of the most probable path to

some other state i at time $t - 1$, followed by a transition from i to j . Hence

$$\delta_t(j) = \lambda_t(j) \left[\max_i \delta_{t-1}(i) A_{i,j} \right] \quad (9.43)$$

We initialize by setting $\delta_1(j) = \pi_j \lambda_1(j)$.

We often work in the log domain to avoid numerical issues. Let $\delta'_t(j) = -\log \delta_t(j)$, $\lambda'_t(j) = -\log p(\mathbf{y}_t | \mathbf{z}_t = j)$, $A'(i, j) = -\log p(\mathbf{z}_t = j | \mathbf{z}_{t-1} = i)$. Then we have

$$\delta'_t(j) = \lambda'_t(j) + \left[\min_i \delta'_{t-1}(i) + A'(i, j) \right] \quad (9.44)$$

We also need to keep track of the most likely previous (**ancestor**) state, for each possible state that we end up in:

$$a_t(j) \triangleq \operatorname{argmax}_i \delta_{t-1}(i) A_{i,j} = \operatorname{argmin}_i \delta'_{t-1}(i) + A'(i, j) \quad (9.45)$$

That is, $a_t(j)$ stores the identity of the previous state on the most probable path to $\mathbf{z}_t = j$. We will see why we need this in Section 9.2.6.2.

9.2.6.2 Backwards pass

In the backwards pass, we compute the most probable sequence of states using a **traceback** procedure, as follows: $\mathbf{z}_t^* = a_{t+1}(\mathbf{z}_{t+1}^*)$, where we initialize using $\mathbf{z}_T^* = \arg \max_i \delta_T(i)$. This is just following the chain of ancestors along the MAP path.

If there is a unique MAP estimate, the above procedure will give the same result as picking $\hat{z}_t = \operatorname{argmax}_j \gamma_t(j)$, computed by forwards-backwards, as shown in [WF01b]. However, if there are multiple posterior modes, the latter approach may not find any of them, since it chooses each state independently, and hence may break ties in a manner that is inconsistent with its neighbors. The traceback procedure avoids this problem, since once \mathbf{z}_t picks its most probable state, the previous nodes condition on this event, and therefore they will break ties consistently.

9.2.6.3 Example

In Figure 9.3(c), we show the Viterbi trace for the casino HMM. We see that, most of the time, the estimated state corresponds to the true state.

In Figure 9.6, we give a detailed worked example of the Viterbi algorithm, based on [Rus+95]. Suppose we observe the sequence of discrete observations $\mathbf{y}_{1:4} = (C_1, C_3, C_4, C_6)$, representing codebook entries in a vector-quantized version of a speech signal. The model starts in state $\mathbf{z}_1 = S_1$. The probability of generating $x_1 = C_1$ in S_1 is 0.5, so we have $\delta_1(1) = 0.5$, and $\delta_1(i) = 0$ for all other states. Next we can self-transition to S_1 with probability 0.3, or transition to S_2 with probability 0.7. If we end up in S_1 , the probability of generating $x_2 = C_3$ is 0.3; if we end up in S_2 , the probability of generating $x_2 = C_3$ is 0.2. Hence we have

$$\delta_2(1) = \delta_1(1) A(1, 1) \lambda_2(1) = 0.5 \cdot 0.3 \cdot 0.3 = 0.045 \quad (9.46)$$

$$\delta_2(2) = \delta_1(1) A(1, 2) \lambda_2(2) = 0.5 \cdot 0.7 \cdot 0.2 = 0.07 \quad (9.47)$$

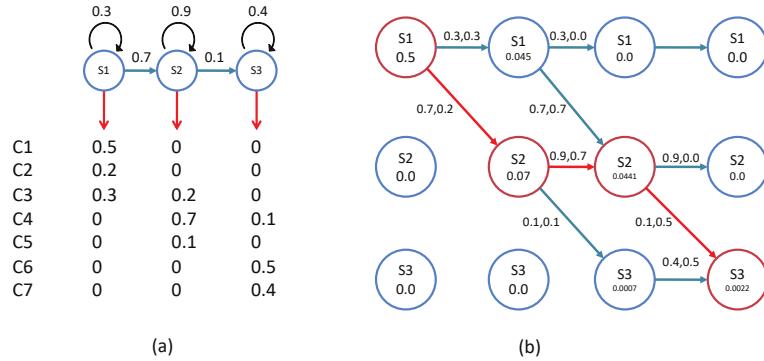


Figure 9.6: Illustration of Viterbi decoding in a simple HMM for speech recognition. (a) A 3-state HMM for a single phone. We are visualizing the state transition diagram. We assume the observations have been vector quantized into 7 possible symbols, C_1, \dots, C_7 . Each state S_1, S_2, S_3 has a different distribution over these symbols. Adapted from Figure 15.20 of [RN02]. (b) Illustration of the Viterbi algorithm applied to this model, with data sequence C_1, C_3, C_4, C_6 . The columns represent time, and the rows represent states. The numbers inside the circles represent the $\delta_t(j)$ value for that state. An arrow from state i at $t - 1$ to state j at t is annotated with two numbers: the first is the probability of the $i \rightarrow j$ transition, and the second is the probability of generating observation y_t from state j . The red lines/circles represent the most probable sequence of states. Adapted from Figure 24.27 of [RN95].

Thus state 2 is more probable at $t = 2$; see the second column of Figure 9.6(b). The algorithm continues in this way until we have reached the end of the sequence. Once we have reached the end, we can follow the red arrows back to recover the MAP path (which is 1,2,2,3).

For more details on HMMs for automatic speech recognition (ASR) see e.g., [JM08].

9.2.6.4 Time and space complexity

The time complexity of Viterbi is clearly $O(K^2T)$ in general, and the space complexity is $O(KT)$, both the same as forwards-backwards. If the transition matrix has the form $A_{i,j} \propto \rho(\mathbf{z}_j - \mathbf{z}_i)$, where \mathbf{z}_i is the continuous vector represented by state i and $\rho(u)$ is some scalar cost function, such as Euclidean distance, we can implement Viterbi in $O(TK)$ time, by using the generalized distance transform to implement Equation (9.44). See [FHK03; FH12] for details.

9.2.6.5 N-best list

There are often multiple paths which have the same likelihood. The Viterbi algorithm returns one of them, but can be extended to return the top N paths [SC90; NG01]. This is called the **N-best list**. Computing such a list can provide a better summary of the posterior uncertainty.

In addition, we can perform **discriminative reranking** [CK05] of all the sequences in \mathcal{L}_N , based on global features derived from $(\mathbf{y}_{1:T}, \mathbf{z}_{1:T})$. This technique is widely used in speech recognition. For example, consider the sentence “recognize speech”. It is possible that the most probable interpretation by the system of this acoustic signal is “wreck a nice speech”, or maybe “wreck a nice beach” (see

Figure 34.3). Maybe the correct interpretation is much lower down on the list. However, by using a re-ranking system, we may be able to improve the score of the correct interpretation based on a more global context.

One problem with the N -best list is that often the top N paths are very similar to each other, rather than representing qualitatively different interpretations of the data. Instead we might want to generate a more diverse set of paths to more accurately represent posterior uncertainty. One way to do this is to sample paths from the posterior, as we discuss in Section 9.2.7. Another way is to use a determinantal point process (Supplementary Section 31.8.5) which encourages points to be diverse [Bat+12; ZA12].

9.2.7 Forwards filtering backwards sampling

Rather than computing the single most probable path, it is often useful to sample multiple paths from the posterior: $\mathbf{z}_{1:T}^s \sim p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T})$. We can do this by modifying the forwards filtering backwards smoothing algorithm from Section 9.2.4, so that we draw samples on the backwards pass, rather than computing marginals. This is called **forwards filtering backwards sampling** (also sometimes unfortunately abbreviated to FFBS). In particular, note that we can write the joint from right to left using

$$p(\mathbf{z}_{1:T} | \mathbf{y}_{1:T}) = p(\mathbf{z}_T | \mathbf{y}_{1:T}) p(\mathbf{z}_{T-1} | \mathbf{z}_T, \mathbf{y}_{1:T}) p(\mathbf{z}_{T-2} | \mathbf{z}_{T-1}, \mathbf{z}_{T-2}, \mathbf{y}_{1:T}) \cdots p(\mathbf{z}_1 | \mathbf{z}_2, \mathbf{z}_{3:T}, \mathbf{y}_{1:T}) \quad (9.48)$$

$$= p(\mathbf{z}_T | \mathbf{y}_{1:T}) \prod_{t=T-1}^1 p(\mathbf{z}_t | \mathbf{z}_{t+1}, \mathbf{y}_{1:T}) \quad (9.49)$$

Thus at step t we sample \mathbf{z}_t^s from $p(\mathbf{z}_t | \mathbf{z}_{t+1}^s, \mathbf{y}_{1:T})$ given in Equation (9.49).

9.3 Belief propagation on trees

The forwards-backwards algorithm for HMMs discussed in Section 9.2.3 (and the Kalman smoother algorithm for LDS which we discuss in Section 8.2.3) can be interpreted as a message passing algorithm applied to a chain structured graphical model. In this section, we generalize these algorithms to work with trees.

9.3.1 Directed vs undirected trees

Consider a pairwise *undirected* graphical model, which can be written as follows:

$$p^*(\mathbf{z}) \triangleq p(\mathbf{z} | \mathbf{y}) \propto \prod_{s \in \mathcal{V}} \psi_s(z_s | \mathbf{y}_s) \prod_{(s,t) \in \mathcal{E}} \psi_{s,t}(z_s, z_t) \quad (9.50)$$

where $\psi_{s,t}(z_s, z_t)$ are the pairwise clique potential, one per edge, $\psi_s(z_s | \mathbf{y}_s)$ are the local evidence potentials, one per node, \mathcal{V} is the set of nodes, and \mathcal{E} is the set of edges. (We will henceforth drop the conditioning on the observed values \mathbf{y} for brevity.)

Now suppose the corresponding graph structure is a tree, such as the one in Figure 9.7a. We can always convert this into a directed tree by picking an arbitrary node as the root, and then “picking

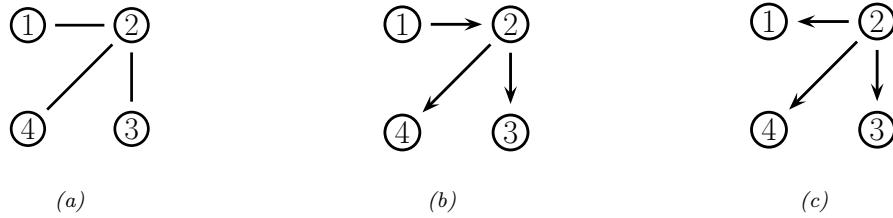


Figure 9.7: An undirected tree and two equivalent directed trees.

the tree up by the root” and orienting all the edges away from the root. For example, if we pick node 1 as the root we get Figure 9.7b. This corresponds to the following directed graphical model:

$$p^*(z) \propto p^*(z_1)p^*(z_2|z_1)p^*(z_3|z_2)p^*(z_4|z_2) \quad (9.51)$$

However, if we pick node 2 as the root, we get Figure 9.7c. This corresponds to the following directed graphical model:

$$p^*(z) \propto p^*(z_2)p^*(z_1|z_2)p^*(z_3|z_2)p^*(z_4|z_2) \quad (9.52)$$

Since these graphs express the same conditional independence properties, they represent the same family of probability distributions, and hence we are free to use any of these parameterizations.

To make the model more symmetric, it is preferable to use an undirected tree. If we define the potentials as (possibly unnormalized) marginals (i.e., $\psi_s(z_s) \propto p^*(z_s)$ and $\psi_{s,t}(z_s, z_t) = p^*(z_s, z_t)$), then we can write

$$p^*(z) \propto \prod_{s \in \mathcal{V}} p^*(z_s) \prod_{(s,t) \in \mathcal{E}} \frac{p^*(z_s, z_t)}{p^*(z_s)p^*(z_t)} \quad (9.53)$$

For example, for Figure 9.7a we have

$$p^*(z_1, z_2, z_3, z_4) \propto p^*(z_1)p^*(z_2)p^*(z_3)p^*(z_4) \frac{p^*(z_1, z_2)p^*(z_2, z_3)p^*(z_2, z_4)}{p^*(z_1)p^*(z_2)p^*(z_3)p^*(z_4)p^*(z_2)p^*(z_4)} \quad (9.54)$$

To see the equivalence with the directed representation, we can cancel terms to get

$$p^*(z_1, z_2, z_3, z_4) \propto p^*(z_1, z_2) \frac{p^*(z_2, z_3)}{p^*(z_2)} \frac{p^*(z_2, z_4)}{p^*(z_2)} \quad (9.55)$$

$$= p^*(z_1)p^*(z_2|z_1)p^*(z_3|z_2)p^*(z_4|z_2) \quad (9.56)$$

$$= p^*(z_2)p^*(z_1|z_2)p^*(z_3|z_2)p^*(z_4|z_2) \quad (9.57)$$

where $p^*(z_t|z_s) = p^*(z_s, z_t)/p^*(z_s)$.

Thus a tree can be represented as either an undirected or directed graph. Both representations can be useful, as we will see.

```

// Collect to root
for each node s in post-order
    bels(zs) ∝ ψs(zs) ∏t ∈ chs mt → s(zs)
    t = parent(s)
    ms → t(zt) = ∑zs ψst(zs, zt)bels(zs)

// Distribute from root
for each node t in pre-order
    s = parent(t)
    ms → t(zt) = ∑zs ψst(zs, zt) ∙ bels(zs)
    belt(zt) ∝ belt(zt)ms → t(zt)

```

Figure 9.8: Belief propagation on a pairwise, rooted tree.

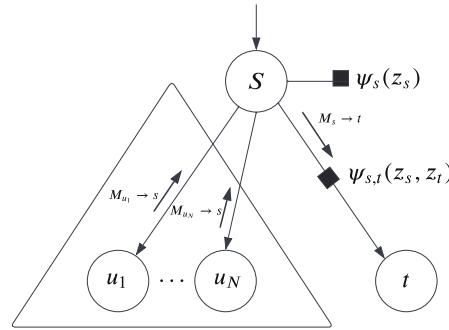


Figure 9.9: Illustration of how the top-down message from s to t is computed during BP on a tree. The u_i nodes are the other children of s , besides t . Square nodes represent clique potentials.

9.3.2 Sum-product algorithm

In this section, we assume that our model is an undirected tree, as in Equation (9.50). However, we will pick an arbitrary node as a root, and orient all the edges downwards away from this root, so that each node has a unique parent. For a directed, rooted tree, we can compute various node orderings. In particular, in a **pre-order**, we traverse from the root to the left subtree and then to right subtree, top to bottom. In a **post-order**, we traverse from the left subtree to the right subtree and then to the root, bottom to top. We will use both of these below.

We now present the **sum-product algorithm** for trees. We first send messages from the leaves to the root. This is the generalization of the forwards pass from Section 9.2.2. Let $m_{s \rightarrow t}(z_t)$ denote the message from node s to node t . This summarizes the belief state about z_t given all the evidence in the tree below the $s - t$ edge. Consider a node s in the ordering. We update its belief state by

combining the incoming messages from all its children with its own local evidence:

$$\text{bel}_s(z_s) \propto \psi_s(z_s) \prod_{t \in \text{ch}_s} m_{t \rightarrow s}(z_s) \quad (9.58)$$

To compute the outgoing message that s should send to its parent t , we pass the local belief through the pairwise potential linking s and t , and then marginalize out s to get

$$m_{s \rightarrow t}(z_t) = \sum_{z_s} \psi_{st}(z_s, z_t) \text{bel}_s(z_s) \quad (9.59)$$

At the root of the tree, $\text{bel}_t(z_t) = p(z_t | \mathbf{y})$ will have seen all the evidence. It can then send messages back down to the leaves. The message that s sends to its child t should be the product of all the messages that s received from all its *other* children u , passed through the pairwise potential, and then marginalized:

$$m_{s \rightarrow t}(z_t) = \sum_{z_t} \left(\psi_s(z_s) \psi_{st}(z_s, z_t) \prod_{u \in \text{ch}_s \setminus t} m_{u \rightarrow s}(z_s) \right) \quad (9.60)$$

See Figure 9.9. Instead of multiplying all-but-one of the messages that s has received, we can multiply all of them and then divide out by the $t \rightarrow s$ message from child t . The advantage of this is that the product of all the messages has already been computed in Equation (9.58), so we don't need to recompute that term. Thus we get

$$m_{s \rightarrow t}(z_t) = \sum_{z_s} \psi_{st}(z_s, z_t) \frac{\text{bel}_s(z_s)}{m_{t \rightarrow s}(z_s)} \quad (9.61)$$

We can think of $\text{bel}_s(z_s)$ as the new updated posterior $p(z_s | \mathbf{y})$ given all the evidence, and $m_{t \rightarrow s}(z_s)$ as the prior predictive $p(z_s | \mathbf{y}_t^-)$, where \mathbf{y}_t^- is all the evidence in the subtree rooted at t . Thus the ratio contains the new evidence that t did not already know about from its own subtree. We use this to update the belief state at node t to get:

$$\text{bel}_t(z_t) \propto \text{bel}_t(z_t) m_{s \rightarrow t}(z_t) \quad (9.62)$$

(Note that Equation (9.58) is a special case of this where we don't divide out by $m_{s \rightarrow t}$, since in the upwards pass, there is no incoming message from the parent.) This is analogous to the backwards smoothing equation in Equation (9.37), with $\alpha_t(i)$ replaced by $\text{bel}_t(z_t = i)$, $A(i, j)$ replaced by $\psi_{st}(z_s = i, z_t = j)$, $\gamma_{t+1}(j)$ replaced by $\text{bel}_s(z_s = j)$, and $\alpha_{t+1|t}(j)$ replaced by $m_{t \rightarrow s}(z_s = j)$.

See Figure 9.8 for the overall pseudocode. This can be generalized to directed trees with multiple root nodes (known as **polytrees**) as described in Supplementary Section 9.1.1.

9.3.3 Max-product algorithm

In Section 9.3.2 we described the sum-product algorithm, that computes the posterior marginals:

$$\text{bel}_i(k) = \gamma_i(k) = p(z_i = k | \mathbf{y}) = \sum_{\mathbf{z}_{-i}} p(z_i = k, \mathbf{z}_{-i} | \mathbf{y}) \quad (9.63)$$

We can replace the sum operation with the max operation to get **max-product belief propagation**. The result of this computation are a set of **max marginals** for each node:

$$\zeta_i(k) = \max_{\mathbf{z}_{-i}} p(z_i = k, \mathbf{z}_{-i} | \mathbf{y}) \quad (9.64)$$

We can derive two different kinds of “MAP” estimates from these local quantities. The first is $\hat{\mathbf{z}}_i = \text{argmax}_k \gamma_i(k)$; this is known as the **maximizer of the posterior marginal** or MPM estimate (see e.g., [MMP87; SM12]); let $\hat{\mathbf{z}} = [\hat{z}_1, \dots, \hat{z}_{N_z}]$ be the sequence of such estimates. The second is $\tilde{\mathbf{z}}_i = \text{argmax}_k \zeta_i(k)$; we call this the **maximizer of the max marginal** or MMM estimate; let $\tilde{\mathbf{z}} = [\tilde{z}_1, \dots, \tilde{z}_{N_z}]$.

An interesting question is: what, if anything, do these estimates have to do with the “true” MAP estimate, $\mathbf{z}^* = \text{argmax}_{\mathbf{z}} p(\mathbf{z} | \mathbf{y})$? We discuss this below.

9.3.3.1 Connection between MMM and MAP

In [YW04], they showed that, if the max marginals are unique and computed exactly (e.g., if the graph is a tree), then $\tilde{\mathbf{z}} = \mathbf{z}^*$. This means we can recover the global MAP estimate by running max product BP and then setting each node to its local max (i.e., using the MMM estimate).

However, if there are ties in the max marginals (corresponding to the case where there is more than one globally optimal solution), this “local stitching” process may result in global inconsistencies.

If we have a tree-structured model, we can use a **traceback** procedure, analogous to the Viterbi algorithm (Section 9.2.6), in which we clamp nodes to their optimal values while working backwards from the root. For details, see e.g., [KF09a, p569].

Unfortunately, traceback does not work on general graphs. An alternative, iterative approach, proposed in [YW04], is follows. First we run max product BP, and clamp all nodes which have unique max marginals to their optimal values; we then clamp a single ambiguous node to an optimal value, and condition on all these clamped values as extra evidence, and perform more rounds of message passing, until all ties are broken. This may require many rounds of inference, although the number of non-clamped (hidden) variables gets reduced at each round.

9.3.3.2 Connection between MPM and MAP

In this section, we discuss the MPM estimate, $\hat{\mathbf{z}}$, which computes the maximum of the posterior marginals. In general, this does not correspond to the MAP estimate, even if the posterior marginals are exact. To see why, note that MPM just looks at the belief state for each node given all the visible evidence, but ignores any dependencies or constraints that might exist in the prior.

To illustrate why this could be a problem, consider the error correcting code example from Section 5.5, where we defined $p(\mathbf{z}, \mathbf{y}) = p(z_1)p(z_2)p(z_3|z_1, z_2)\prod_{i=1}^3 p(y_i|z_i)$, where all variables are binary. The priors $p(z_1)$ and $p(z_2)$ are uniform. The conditional term $p(z_3|z_1, z_2)$ is deterministic, and computes the parity of (z_1, z_2) . In particular, we have $p(z_3 = 1|z_1, z_2) = \mathbb{I}(\text{odd}(z_1, z_2))$, so that the total number of 1s in the block $\mathbf{z}_{1:3}$ is even. The likelihood terms $p(y_i|z_i)$ represent a bit flipping noisy channel model, with noise level $\alpha = 0.2$.

Suppose we observe $\mathbf{y} = (1, 0, 0)$. In this case, the exact posterior marginals are as follows:² $\gamma_1 = [0.3469, 0.6531]$, $\gamma_2 = [0.6531, 0.3469]$, $\gamma_3 = [0.6531, 0.3469]$. The exact max marginals are all the same,

2. See [error_correcting_code_demo.ipynb](#) for the code.

namely $\zeta_i = [0.3265, 0.3265]$. Finally, the 3 global MAP estimates are $\mathbf{z}^* \in \{[0, 0, 0], [1, 1, 0], [1, 0, 1]\}$, each of which corresponds to a single bit flip from the observed vector. The MAP estimates are all valid code words (they have an even number of 1s), and hence are sensible hypotheses about the value of \mathbf{z} . By contrast, the MPM estimate is $\hat{\mathbf{z}} = [1, 0, 0]$, which is not a legal codeword. (And in this example, the MMM estimate is not well defined, since the max marginals are not unique.)

So, which method is better? This depends on our loss function, as we discuss in Section 34.1. If we want to minimize the prediction error of each z_i , also called **bit error**, we should compute the MPM. If we want to minimize the prediction error for the entire sequence \mathbf{z} , also called **word error**, we should use MAP, since this can take global constraints into account.

For example, suppose we are performing speech recognition and someone says “recognize speech”. MPM decoding may return “wreck a nice beach”, since locally it may be that “beach” is the most probable interpretation of “speech” when viewed in isolation (see Figure 34.3). However, MAP decoding would infer that “recognize speech” is the more likely overall interpretation, by taking into account the language model prior, $p(\mathbf{z})$.

On the other hand, if we don’t have strong constraints, the MPM estimate can be more robust [MMP87; SM12], since it marginalizes out the other nodes, rather than maxing them out. For example, in the casino HMM example in Figure 9.3, we see that the MPM method makes 49 bit errors (out of a total possible of $T = 300$), and the MAP path makes 60 errors.

9.3.3.3 Connection between MPE and MAP

In the graphical models literature, computing the jointly most likely setting of all the latent variables, $\mathbf{z}^* = \operatorname{argmax}_{\mathbf{z}} p(\mathbf{z}|\mathbf{y})$, is known as the **most probable explanation** or **MPE** [Pea88]. In that literature, the term “MAP” is used to refer to the case where we maximize some of the hidden variables, and marginalize (sum out) the rest. For example, if we maximize a single node, z_i , but sum out all the others, \mathbf{z}_{-i} , we get the MPM $\hat{z}_i = \operatorname{argmax}_{z_i} \sum_{\mathbf{z}_{-i}} p(\mathbf{z}|\mathbf{y})$.

We can generalize the MPM estimate to compute the best guess for a set of query variables Q , given evidence on a set of visible variables V , marginalizing out the remaining variables R , to get

$$\mathbf{z}_Q^* = \arg \max_{\mathbf{z}_Q} \sum_{\mathbf{z}_R} p(\mathbf{z}_Q, \mathbf{z}_R | \mathbf{z}_V) \quad (9.65)$$

(Here \mathbf{z}_R are called **nuisance variables**, since they are not of interest, and are not observed.) In [Pea88], this is called a MAP estimate, but we will call it an MPM estimate, to avoid confusion with the ML usage of the term “MAP” (where we maximize everything jointly).

9.4 Loopy belief propagation

In this section, we extend belief propagation to work on graphs with cycles or loops; this is called **loopy belief propagation** or **LBP**. Unfortunately, this method may not converge, and even if it does, it is not clear if the resulting estimates are valid. Indeed, Judea Pearl, who invented belief propagation for trees, wrote the following about loopy BP in 1988:

When loops are present, the network is no longer singly connected and local propagation schemes will invariably run into trouble ... If we ignore the existence of loops and permit the nodes to continue communicating with each other as if the network were singly connected,

messages may circulate indefinitely around the loops and the process may not converge to a stable equilibrium ... Such oscillations do not normally occur in probabilistic networks ... which tend to bring all messages to some stable equilibrium as time goes on. However, this asymptotic equilibrium is not coherent, in the sense that it does not represent the posterior probabilities of all nodes of the network. — [Pea88, p.195]

Despite these reservations, Pearl advocated the use of belief propagation in loopy networks as an approximation scheme (J. Pearl, personal communication). [MWJ99] found empirically that it works on various graphical models, and it is now used in many real world applications, some of which we discuss below. In addition, there is now some theory justifying its use in certain cases, as we discuss below. (For more details, see e.g., [Yed11].)

9.4.1 Loopy BP for pairwise undirected graphs

In this section, we assume (for notational simplicity) that our model is an undirected pairwise PGM, as in Equation (9.50). However, unlike Section 9.3.2, we do not assume the graph is a tree. We can apply the same message passing equations as before. However, since there is no natural node ordering, we will do this in a parallel, asynchronous way. The basic idea is that all nodes receive messages from their neighbors in parallel, they then update their belief states, and finally they send new messages back out to their neighbors. This message passing process repeats until convergence. This kind of computing architecture is called a **systolic array**, due to its resemblance to a beating heart.

More precisely, we initialize all messages to the all 1's vector. Then, in parallel, each node absorbs messages from all its neighbors using

$$\text{bel}_s(z_s) \propto \psi_s(z_s) \prod_{t \in \text{nbr}_s} m_{t \rightarrow s}(z_s) \quad (9.66)$$

Then, in parallel, each node sends messages to each of its neighbors:

$$m_{s \rightarrow t}(z_t) = \sum_{z_s} \left(\psi_s(z_s) \psi_{st}(z_s, z_t) \prod_{u \in \text{nbr}_s \setminus t} m_{u \rightarrow s}(z_s) \right) \quad (9.67)$$

The $m_{s \rightarrow t}$ message is computed by multiplying together all incoming messages, except the one sent by the recipient, and then passing through the ψ_{st} potential. We continue this process until convergence. If the graph is a tree, the method is guaranteed to converge after $D(G)$ iterations, where $D(G)$ is the **diameter** of the graph, that is, the largest distance between any two nodes.

9.4.2 Loopy BP for factor graphs

To implement loopy BP for general graphs, including those with higher-order clique potentials (beyond pairwise), it is useful to use a factor graph representation described in Section 4.6.1. In this section, we summarize the BP equations for the bipartite version of factor graphs, as derived in [KFL01].³ For a version that works for Forney factor graphs, see [Loe+07].

3. For an efficient JAX implementation of these equations for discrete factor graphs, see <https://github.com/deepmind/PGMax>. For the Gaussian case, see <https://github.com/probml/pgm-jax>.

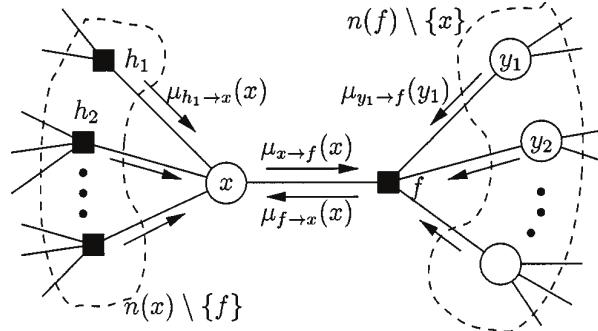


Figure 9.10: Message passing on a bipartite factor graph. Square nodes represent factors, and circles represent variables. The y_i nodes correspond to the neighbors x'_i of f other than x . From Figure 6 of [KFL01]. Used with kind permission of Brendan Frey.

In the case of bipartite factor graphs, we have two kinds of messages: variables to factors

$$m_{x \rightarrow f}(x) = \prod_{h \in \text{nbr}(x) \setminus \{f\}} m_{h \rightarrow x}(x) \quad (9.68)$$

and factors to variables

$$m_{f \rightarrow x}(x) = \sum_{x'} f(x, x') \prod_{x' \in \text{nbr}(f) \setminus \{x\}} m_{x' \rightarrow f}(x') \quad (9.69)$$

Here $\text{nbr}(x)$ are all the factors that are connected to variable x , and $\text{nbr}(f)$ are all the variables that are connected to factor f . These messages are illustrated in Figure 9.10. At convergence, we can compute the final beliefs as a product of incoming messages:

$$\text{bel}(x) \propto \prod_{f \in \text{nbr}(x)} m_{f \rightarrow x}(x) \quad (9.70)$$

The order in which the messages are sent can be determined using various heuristics, such as computing a spanning tree, and picking an arbitrary node as root. Alternatively, the update ordering can be chosen adaptively using **residual belief propagation** [EMK06]. Or fully parallel, asynchronous implementations can be used.

9.4.3 Gaussian belief propagation

It is possible to generalize (loopy) belief propagation to the Gaussian case, by using the ‘‘calculus for linear Gaussian models’’ in Section 2.3.3 to compute the messages and beliefs. Note that computing the posterior mean in a linear-Gaussian system is equivalent to solving a linear system, so these methods are also useful for linear algebra. See e.g., [PL03; Bic09; Du+18] for details.

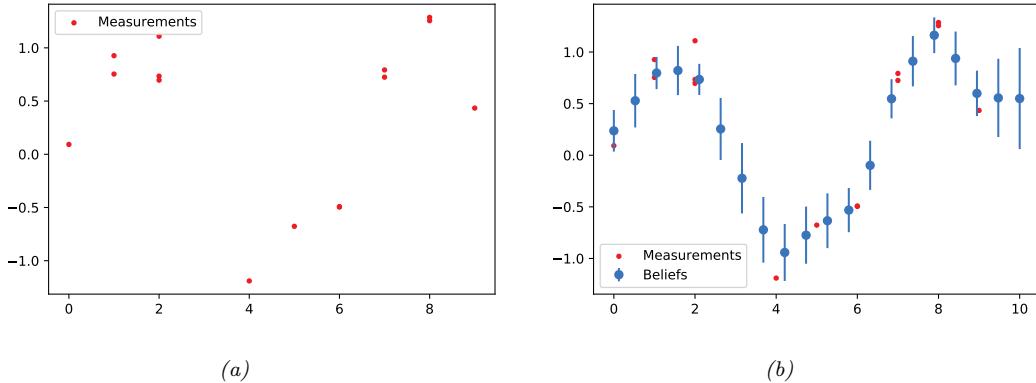


Figure 9.11: Interpolating noisy data using Gaussian belief propagation applied to a 1d MRF. Generated by [gauss-bp-1d-line.ipynb](#).

As an example of Gaussian BP, consider the problem of interpolating noisy data in 1d, as discussed in [OED21]. In particular, let $f : \mathbb{R} \rightarrow \mathbb{R}$ be an unknown function for which we get N noisy measurements y_i at locations x_i . We want to estimate $z_i = f(g_i)$ at G grid locations g_i . Let x_i be the closest location to g_i . Then we assume the measurement factor is as follows:

$$\psi_i(z_{i-1}, z_i) = \frac{1}{\sigma^2}(\hat{y}_i - y_i)^2 \quad (9.71)$$

$$\hat{y}_i = (1 - \gamma_i)z_{i-1} + \gamma_i z_i \quad (9.72)$$

$$\gamma_i = \frac{x_i - g_i}{g_i - g_{i-1}} \quad (9.73)$$

Here \hat{y}_i is the predicted measurement. The potential function makes the unknown function values z_{i-1} and z_i move closer to the observation, based on how far these grid points are from where the measurement was taken. In addition, we add a pairwise smoothness potential, that encodes the prior that z_i should be close to z_{i-1} and z_{i+1} :

$$\phi_i(z_{i-1}, z_i) = \frac{1}{\tau^2} \delta_i^2 \quad (9.74)$$

$$\delta_i = z_i - z_{i-1} \quad (9.75)$$

The overall model is

$$p(\mathbf{z}|\mathbf{x}, \mathbf{y}, \mathbf{g}, \sigma^2, \tau^2) \propto \prod_{i=1}^G \psi_i(z_{i-1}, z_i) \phi_i(z_{i-1}, z_i) \quad (9.76)$$

Suppose the true underlying function is a sine wave. We show some sample data in Figure 9.11(a). We then apply Gaussian BP. Since this model is a chain, and the model is linear-Gaussian, the resulting posterior marginals, shown in Figure 9.11(b), are exact. We see that the method has inferred the underlying sine shape just based on a smoothness prior.

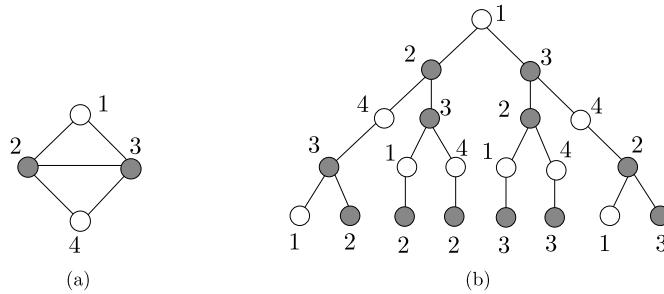


Figure 9.12: (a) A simple loopy graph. (b) The computation tree, rooted at node 1, after 4 rounds of message passing. Nodes 2 and 3 occur more often in the tree because they have higher degree than nodes 1 and 2. From Figure 8.2 of [WJ08]. Used with kind permission of Martin Wainwright.

To perform message passing in models with non-linear (but Gaussian) potentials, we can generalize the extended Kalman filter techniques from Section 8.3.2 and the moment matching techniques (based on quadrature/sigma points) from Section 8.5.1 and Section 8.5.1.1 from chains to general factor graphs (see e.g., [MHH14; PHR18; HPR19]). To extend to the non-Gaussian case, we can use **non-parametric BP** or **particle BP** (see e.g., [Sud+03; Isa03; Sud+10; Pac+14]), which uses ideas from particle filtering (Section 13.2).

9.4.4 Convergence

Loopy BP may not converge, or may only converge slowly. In this section, we discuss some techniques that increase the chances of convergence, and the speed of convergence.

9.4.4.1 When will LBP converge?

The details of the analysis of when LBP will converge are beyond the scope of this chapter, but we briefly sketch the basic idea. The key analysis tool is the **computation tree**, which visualizes the messages that are passed as the algorithm proceeds. Figure 9.12 gives a simple example. In the first iteration, node 1 receives messages from nodes 2 and 3. In the second iteration, it receives one message from node 3 (via node 2), one from node 2 (via node 3), and two messages from node 4 (via nodes 2 and 3). And so on.

The key insight is that T iterations of LBP is equivalent to exact computation in a computation tree of height $T + 1$. If the strengths of the connections on the edges is sufficiently weak, then the influence of the leaves on the root will diminish over time, and convergence will occur. See [MK05; WJ08] and references therein for more information.

9.4.4.2 Making LBP converge

Although the theoretical convergence analysis is very interesting, in practice, when faced with a model where LBP is not converging, what should we do?

One simple way to increase the chance of convergence is to use **damping**. That is, at iteration k ,

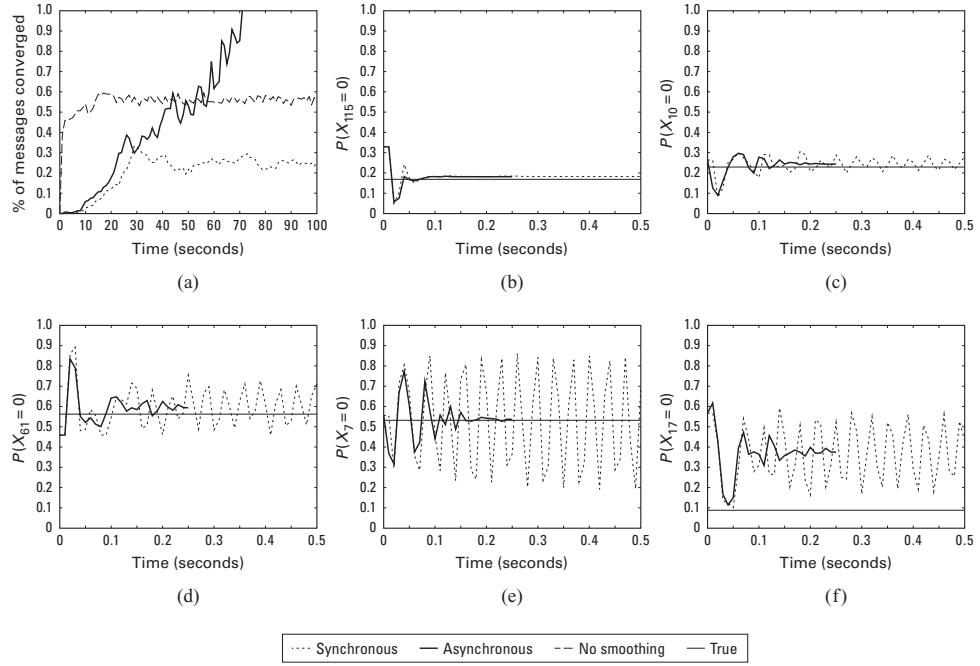


Figure 9.13: Illustration of the behavior of loopy belief propagation on an 11×11 Ising grid with random potentials, $w_{ij} \sim \text{Unif}(-C, C)$, where $C = 11$. For larger C , inference becomes harder. (a) Percentage of messages that have converged vs time for 3 different update schedules: Dotted = damped synchronous (few nodes converge), dashed = undamped asynchronous (half the nodes converge), solid = damped asynchronous (all nodes converge). (b-f) Marginal beliefs of certain nodes vs time. Solid straight line = truth, dashed = synchronous, solid = damped asynchronous. From Figure 11.C.1 of [KF09a]. Used with kind permission of Daphne Koller.

we use an update of the form

$$m_{t \rightarrow s}^k(x_s) = \lambda m_{t \rightarrow s}(x_s) + (1 - \lambda) m_{t \rightarrow s}^{k-1}(x_s) \quad (9.77)$$

where $m_{t \rightarrow s}(x_s)$ is the standard undamped message, where $0 \leq \lambda \leq 1$ is the damping factor. Clearly if $\lambda = 1$ this reduces to the standard scheme, but for $\lambda < 1$, this partial updating scheme can help improve convergence. Using a value such as $\lambda \sim 0.5$ is standard practice. The benefits of this approach are shown in Figure 9.13, where we see that damped updating results in convergence much more often than undamped updating (see [ZLG20] for some analysis of the benefits of damping).

It is possible to devise methods, known as **double loop algorithms**, which are guaranteed to converge to a local minimum of the same objective that LBP is minimizing [Yui01; WT01]. Unfortunately, these methods are rather slow and complicated, and the accuracy of the resulting marginals is usually not much greater than with standard LBP. (Indeed, oscillating marginals is sometimes a sign that the LBP approximation itself is a poor one.) Consequently, these techniques are not very widely used (although see [GF21] for a newer technique).

9.4.4.3 Increasing the convergence rate with adaptive scheduling

The standard approach when implementing LBP is to perform **synchronous updates**, where all nodes absorb messages in parallel, and then send out messages in parallel. That is, the new messages at iteration $k + 1$ are computed in parallel using

$$\mathbf{m}_{1:E}^{k+1} = (f_1(\mathbf{m}^k), \dots, f_E(\mathbf{m}^k)) \quad (9.78)$$

where E is the number of edges, and $f_i(\mathbf{m})$ is the function that computes the message for edge i given all the old messages. This is analogous to the Jacobi method for solving linear systems of equations.

It is well known [Ber97b] that the Gauss-Seidel method, which performs **asynchronous updates** in a fixed round-robin fashion, converges faster when solving linear systems of equations. We can apply the same idea to LBP, using updates of the form

$$\mathbf{m}_i^{k+1} = f_i(\{\mathbf{m}_j^{k+1} : j < i\}, \{\mathbf{m}_j^k : j > i\}) \quad (9.79)$$

where the message for edge i is computed using new messages (iteration $k + 1$) from edges earlier in the ordering, and using old messages (iteration k) from edges later in the ordering.

This raises the question of what order to update the messages in. One simple idea is to use a fixed or random order. The benefits of this approach are shown in Figure 9.13, where we see that (damped) asynchronous updating results in convergence much more often than synchronous updating.

However, we can do even better by using an adaptive ordering. The intuition is that we should focus our computational efforts on those variables that are most uncertain. [EMK06] proposed a technique known as **residual belief propagation**, in which messages are scheduled to be sent according to the norm of the difference from their previous value. That is, we define the residual of new message $m_{s \rightarrow t}$ at iteration k to be

$$r(s, t, k) = \|\log m_{s \rightarrow t} - \log m_{s \rightarrow t}^k\|_\infty = \max_j |\log \frac{m_{s \rightarrow t}(j)}{m_{s \rightarrow t}^k(j)}| \quad (9.80)$$

We can store messages in a priority queue, and always send the one with highest residual. When a message is sent from s to t , all of the other messages that depend on $m_{s \rightarrow t}$ (i.e., messages of the form $m_{t \rightarrow u}$ where $u \in \text{nbr}(t) \setminus s$) need to be recomputed; their residual is recomputed, and they are added back to the queue. In [EMK06], they showed (experimentally) that this method converges more often, and much faster, than using synchronous updating, or asynchronous updating with a fixed order.

A refinement of residual BP was presented in [SM07]. In this paper, they use an upper bound on the residual of a message instead of the actual residual. This means that messages are only computed if they are going to be sent; they are not just computed for the purposes of evaluating the residual. This was observed to be about five times faster than residual BP, although the quality of the final results are similar.

9.4.5 Accuracy

For a graph with a single loop, one can show that the max-product version of LBP will find the correct MAP estimate, if it converges [Wei00]. For more general graphs, one can bound the error in the approximate marginals computed by LBP, as shown in [WJW03; IFW05; Vin+10b].

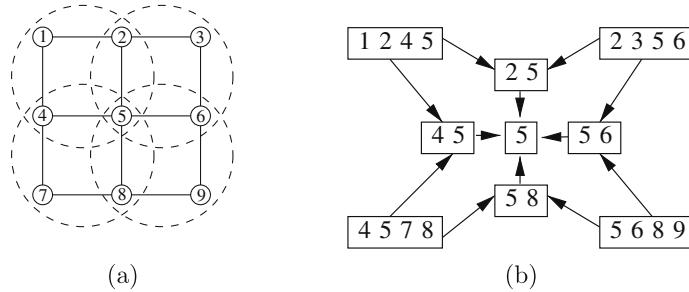


Figure 9.14: (a) Clusters superimposed on a 3×3 lattice graph. (b) Corresponding hyper-graph. Nodes represent clusters, and edges represent set containment. From Figure 4.5 of [WJ08]. Used with kind permission of Martin Wainwright.

Much stronger results are available in the case of Gaussian models. In particular, it can be shown that, if the method converges, the means are exact, although the variances are not (typically the beliefs are over confident). See e.g., [WF01a; JMW06; Bic09; Du+18] for details.

9.4.6 Generalized belief propagation

We can improve the accuracy of loopy BP by clustering together nodes that form a tight loop. This is known as the **cluster variational method**, or **generalized belief propagation** [YFW00].

The result of clustering is a hyper-graph, which is a graph where there are hyper-edges between sets of vertices instead of between single vertices. Note that a junction tree (Section 9.6) is a kind of hyper-graph. We can represent a hyper-graph using a poset (partially ordered set) diagram, where each node represents a hyper-edge, and there is an arrow $e_1 \rightarrow e_2$ if $e_2 \subset e_1$. See Figure 9.14 for an example.

If we allow the size of the largest hyper-edge in the hyper-graph to be as large as the treewidth of the graph, then we can represent the hyper-graph as a tree, and the method will be exact, just as LBP is exact on regular trees (with treewidth 1). In this way, we can define a continuum of approximations, from LBP all the way to exact inference. See [Supplementary Section 10.4.3.3](#) for more information.

9.4.7 Convex BP

In [Supplementary Section 10.4.3](#) we analyze LBP from a variational perspective, and show that the resulting optimization problem, for both standard and generalized BP, is non-convex. However it is possible to create a version of **convex BP**, as we explain in [Supplementary Section 10.4.4](#), which has the advantage that it will always converge.

9.4.8 Application: error correcting codes

LBP was first proposed by Judea Pearl in his 1988 book [Pea88]. He recognized that applying BP to loopy graphs might not work, but recommended it as a heuristic.

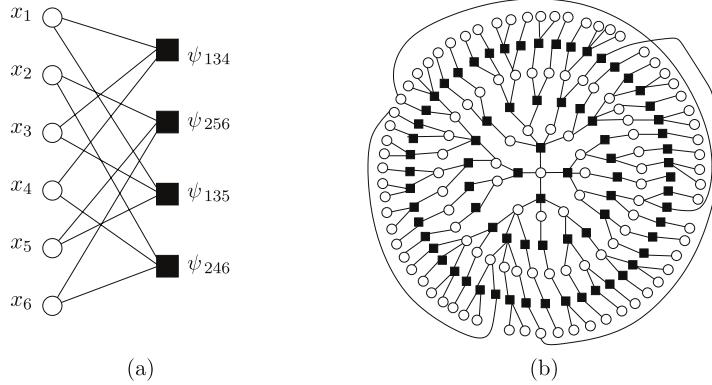


Figure 9.15: (a) A simple factor graph representation of a $(2,3)$ low-density parity check code. Each message bit (hollow round circle) is connected to two parity factors (solid black squares), and each parity factor is connected to three bits. Each parity factor has the form $\psi_{stu}(x_s, x_t, x_u) = \mathbb{I}(x_s \otimes x_t \otimes x_u = 1)$, where \otimes is the xor operator. The local evidence factors for each hidden node are not shown. (b) A larger example of a random LDPC code. We see that this graph is “locally tree-like”, meaning there are no short cycles; rather, each cycle has length $\sim \log m$, where m is the number of nodes. This gives us a hint as to why loopy BP works so well on such graphs. (Note, however, that some error correcting code graphs have short loops, so this is not the full explanation.) From Figure 2.9 from [WJ08]. Used with kind permission of Martin Wainwright.

However, the main impetus behind the interest in LBP arose when McEliece, MacKay, and Cheng [MMC98] showed that a popular algorithm for error correcting codes, known as **turbocodes** [BGT93], could be viewed as an instance of LBP applied to a certain kind of graph.

We introduced error correcting codes in Section 5.5. Recall that the basic idea is to send the source message $\mathbf{x} \in \{0, 1\}^m$ over a noisy channel, and for the receiver to try to infer it given noisy measurements $\mathbf{y} \in \{0, 1\}^m$ or $\mathbf{y} \in \mathbb{R}^m$. That is, the receiver needs to compute $\mathbf{x}^* = \operatorname{argmax}_{\mathbf{x}} p(\mathbf{x}|\mathbf{y}) = \operatorname{argmax}_{\mathbf{x}} \tilde{p}(\mathbf{x})$.

It is standard to represent $\tilde{p}(\mathbf{x})$ as a factor graph (Section 4.6.1), which can easily represent any deterministic relationships (parity constraints) between the bits. A factor graph is a bipartite graph with x_i nodes on one side, and factors on the other. A graph in which each node is connected to n factors, and in which each factor is connected to k nodes, is called an (n, k) code. Figure 9.15(a) shows a simple example of a $(2, 3)$ code, where each bit (hollow round circle) is connected to two parity factors (solid black squares), and each parity factor is connected to three bits. Each parity factor has the form

$$\psi_{stu}(x_s, x_t, x_u) \triangleq \begin{cases} 1 & \text{if } x_s \otimes x_t \otimes x_u = 1 \\ 0 & \text{otherwise} \end{cases} \quad (9.81)$$

If the degrees of the parity checks and variable nodes remain bounded as the blocklength m increases, this is called a **low-density parity check code**, or **LDPC code**. (Turbocodes are constructed in a similar way.)

Figure 9.15(b) shows an example of a randomly constructed LDPC code. This graph is “locally tree-like”, meaning there are no short cycles; rather, each cycle has length $\sim \log m$. This fact is important to the success of LBP, which is only guaranteed to work on tree-structured graphs. Using

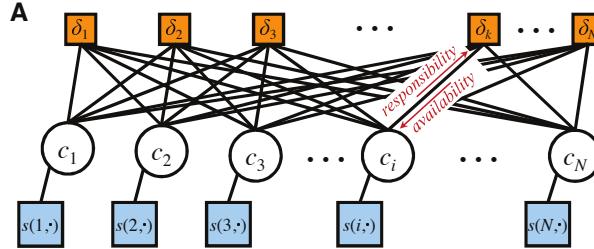


Figure 9.16: Factor graphs for affinity propagation. Circles are variables, squares are factors. Each c_i node has N possible states. From Figure S2 of [FD07a]. Used with kind permission of Brendan Frey.

methods such as these, people have been able to approach the lower bound in Shannon’s channel coding theorem, meaning they have produced codes with very little redundancy for a given amount of noise in the channel. See e.g., [MMC98; Mac03] for more details. Such codes are widely used, e.g., in modern cellphones.

9.4.9 Application: affinity propagation

In this section, we discuss **affinity propagation** [FD07a], which can be seen as an improvement to K-medoids clustering, which takes as input a pairwise similarity matrix. The idea is that each datapoint must choose another datapoint as its exemplar or centroid; some datapoints will choose themselves as centroids, and this will automatically determine the number of clusters. More precisely, let $c_i \in \{1, \dots, N\}$ represent the centroid for datapoint i . The goal is to maximize the following function

$$J(\mathbf{c}) = \sum_{i=1}^N S(i, c_i) + \sum_{k=1}^N \delta_k(\mathbf{c}) \quad (9.82)$$

where $S(i, c_i)$ is the similarity between datapoint i and its centroid c_i . The second term is a penalty term that is $-\infty$ if some datapoint i has chosen k as its exemplar (i.e., $c_i = k$), but k has not chosen itself as an exemplar (i.e., we do not have $c_k = k$). More formally,

$$\delta_k(\mathbf{c}) = \begin{cases} -\infty & \text{if } c_k \neq k \text{ but } \exists i : c_i = k \\ 0 & \text{otherwise} \end{cases} \quad (9.83)$$

This encourages “representative” samples to vote for themselves as centroids, thus encouraging clustering behavior.

The objective function can be represented as a factor graph. We can either use N nodes, each with N possible values, as shown in Figure 9.16, or we can use N^2 binary nodes (see [GF09] for the details). We will assume the former representation.

We can find a strong local maximum of the objective by using max-product loopy belief propagation (Section 9.4). Referring to the model in Figure 9.16, each variable node c_i sends a message to each factor node δ_k . It turns out that this vector of N numbers can be reduced to a scalar message,

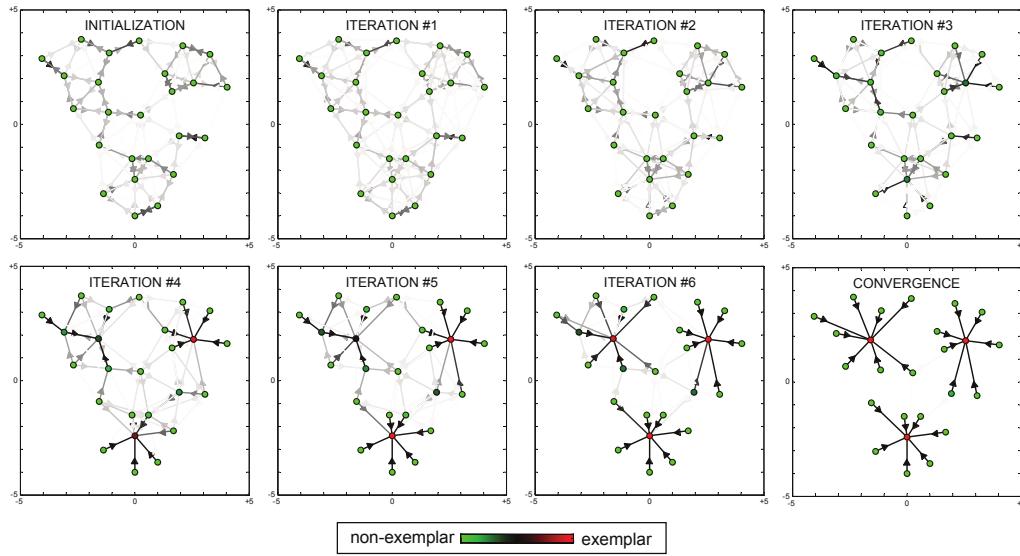


Figure 9.17: Example of affinity propagation. Each point is colored coded by how much it wants to be an exemplar (red is the most, green is the least). This can be computed by summing up all the incoming availability messages and the self-similarity term. The darkness of the $i \rightarrow k$ arrow reflects how much point i wants to belong to exemplar k . From Figure 1 of [FD07a]. Used with kind permission of Brendan Frey.

denoted $r_{i \rightarrow k}$, known as the responsibility. This is a measure of how much i thinks k would make a good exemplar, compared to all the other exemplars i has looked at. In addition, each factor node δ_k sends a message to each variable node c_i . Again this can be reduced to a scalar message, $a_{i \leftarrow k}$, known as the availability. This is a measure of how strongly k believes it should be an exemplar for i , based on all the other datapoints k has looked at.

As usual with loopy BP, the method might oscillate, and convergence is not guaranteed. However, by using damping, the method is very reliable in practice. If the graph is densely connected, message passing takes $O(N^2)$ time, but with sparse similarity matrices, it only takes $O(E)$ time, where E is the number of edges or non-zero entries in S .

The number of clusters can be controlled by scaling the diagonal terms $S(i, i)$, which reflect how much each datapoint wants to be an exemplar. Figure 9.17 gives a simple example of some 2d data, where the negative Euclidean distance was used to measured similarity. The $S(i, i)$ values were set to be the median of all the pairwise similarities. The result is 3 clusters. Many other results are reported in [FD07a], who show that the method significantly outperforms K-medoids.

9.4.10 Emulating BP with graph neural nets

There is a close connection between message passing in PGMs and message passing in graph neural networks (GNNs), which we discuss in Section 16.3.6. However, for PGMs, the message computations are computing using (non-learned) update equations that work for any model; all that is needed

is the graph structure G , model parameters θ , and evidence v . By contrast, GNNs are trained to emulate specific functions using labeled input-output pairs.

It is natural to wonder what happens if we train a GNN on the exact posterior marginals derived from a small PGM, and then apply that trained GNN to a different test PGM. In [Yoo+18; Zha+19d], they show this method can work quite well if the test PGM is similar in structure to the one used for training.

An alternative approach is to start with a known PGM, and then “unroll” the BP message passing algorithm to produce a layered feedforward model, whose connectivity is derived from the graph. The resulting network can then be trained discriminatively for some end-task (not necessarily computing posterior marginals). Thus the BP procedure applied to the PGM just provides a way to design the neural network structure. This method is called **deep unfolding** (see e.g., [HLRW14]), and can often give very good results. (See also [SW20] for a more recent version of this approach, called “**neural enhanced BP**”).

These neural methods are useful if the PGM is fixed, and we want to repeatedly perform inference or prediction with it, using different values of the evidence, but where the set of nodes which are observed is always the same. This is an example of amortized inference, where we train a model to emulate the results of running an iterative optimization scheme (see Section 10.1.5 for more discussion).

9.5 The variable elimination (VE) algorithm

In this section, we discuss an algorithm to compute a posterior marginal $p(z_Q|y)$ for any query set Q , assuming p is defined by a graphical model. Unlike loopy BP, it is guaranteed to give the correct answers even if the graph has cycles. We assume all the hidden nodes are discrete, although a version of the algorithm can be created for the Gaussian case by using the rules for sum and product defined in Section 2.3.3.

9.5.1 Derivation of the algorithm

We will explain the algorithm by applying it to an example. Specifically, we consider the student network from Section 4.2.2.2. Suppose we want to compute $p(J = 1)$, the marginal probability that a person will get a job. Since we have 8 binary variables, we could simply enumerate over all possible assignments to all the variables (except for J), adding up the probability of each joint instantiation:

$$p(J) = \sum_L \sum_S \sum_G \sum_H \sum_I \sum_D \sum_C p(C, D, I, G, S, L, J, H) \quad (9.84)$$

However, this would take $O(2^7)$ time. We can be smarter by **pushing sums inside products**. This is the key idea behind the **variable elimination** algorithm [ZP96], also called **bucket elimination** [Dec96], or, in the context of genetic pedigree trees, the **peeling algorithm** [CTS78].

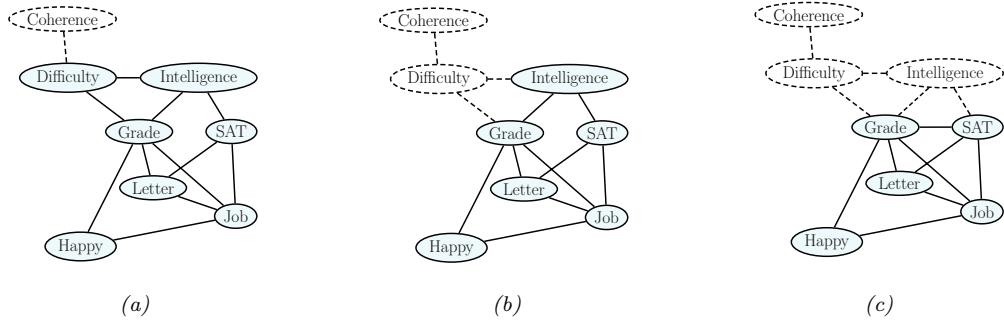


Figure 9.18: Example of the elimination process, in the order C, D, I, H, G, S, L . When we eliminate I (figure c), we add a fill-in edge between G and S , since they are not connected. Adapted from Figure 9.10 of [KF09a].

In our example, we get

$$\begin{aligned}
p(J) &= \sum_{L,S,G,H,I,D,C} p(C, D, I, G, S, L, J, H) \\
&= \sum_{L,S,G,H,I,D,C} \psi_C(C) \psi_D(D, C) \psi_I(I) \psi_G(G, I, D) \psi_S(S, I) \psi_L(L, G) \\
&\quad \times \psi_J(J, L, S) \psi_H(H, G, J) \\
&= \sum_{L,S} \psi_J(J, L, S) \sum_G \psi_L(L, G) \sum_H \psi_H(H, G, J) \sum_I \psi_S(S, I) \psi_I(I) \\
&\quad \times \sum_D \psi_G(G, I, D) \sum_C \psi_C(C) \psi_D(D, C)
\end{aligned}$$

We now evaluate this expression, working right to left as shown in Table 9.1. First we multiply together all the terms in the scope of the \sum_C operator to create the temporary factor

$$\tau'_1(C, D) = \psi_C(C)\psi_D(D, C) \quad (9.85)$$

Then we marginalize out C to get the new factor

$$\tau_1(D) = \sum_C \tau'_1(C, D) \quad (9.86)$$

Next we multiply together all the terms in the scope of the \sum_D operator and then marginalize out to create

$$\tau'_2(G, I, D) = \psi_G(G, I, D)\tau_1(D) \quad (9.87)$$

$$\tau_2(G, I) = \sum_P \tau'_2(G, I, D) \quad (9.88)$$

And so on.

$$\begin{aligned}
& \sum_L \sum_S \psi_J(J, L, S) \sum_G \psi_L(L, G) \sum_H \psi_H(H, G, J) \sum_I \psi_S(S, I) \psi_I(I) \sum_D \psi_G(G, I, D) \underbrace{\sum_C \psi_C(C) \psi_D(D, C)}_{\tau_1(D)} \\
& \sum_L \sum_S \psi_J(J, L, S) \sum_G \psi_L(L, G) \sum_H \psi_H(H, G, J) \sum_I \psi_S(S, I) \psi_I(I) \underbrace{\sum_D \psi_G(G, I, D) \tau_1(D)}_{\tau_2(G, I)} \\
& \sum_L \sum_S \psi_J(J, L, S) \sum_G \psi_L(L, G) \sum_H \psi_H(H, G, J) \underbrace{\sum_I \psi_S(S, I) \psi_I(I) \tau_2(G, I)}_{\tau_3(G, S)} \\
& \sum_L \sum_S \psi_J(J, L, S) \sum_G \psi_L(L, G) \underbrace{\sum_H \psi_H(H, G, J) \tau_3(G, S)}_{\tau_4(G, J)} \\
& \sum_L \sum_S \psi_J(J, L, S) \underbrace{\sum_G \psi_L(L, G) \tau_4(G, J) \tau_3(G, S)}_{\tau_5(J, L, S)} \\
& \sum_L \underbrace{\sum_S \psi_J(J, L, S) \tau_5(J, L, S)}_{\tau_6(J, L)} \\
& \underbrace{\sum_L \tau_6(J, L)}_{\tau_7(J)}
\end{aligned}$$

Table 9.1: Eliminating variables from Figure 4.38 in the order C, D, I, H, G, S, L to compute $P(J)$.

The above technique can be used to compute any marginal of interest, such as $p(J)$ or $p(J, H)$. To compute a conditional, we can take a ratio of two marginals, where the visible variables have been clamped to their known values (and hence don't need to be summed over). For example,

$$p(J = j | I = 1, H = 0) = \frac{p(J = j, I = 1, H = 0)}{\sum_{j'} p(J = j', I = 1, H = 0)} \quad (9.89)$$

9.5.2 Computational complexity of VE

The running time of VE is clearly exponential in the size of the largest factor, since we have to sum over all of the corresponding variables. Some of the factors come from the original model (and are thus unavoidable), but new factors may also be created in the process of summing out. For example, in Table 9.1, we created a factor involving G, I , and S ; but these nodes were not originally present together in any factor.

The order in which we perform the summation is known as the **elimination order**. This can have a large impact on the size of the intermediate factors that are created. For example, consider the ordering in Table 9.1: the largest created factor (beyond the original ones in the model) has size 3, corresponding to $\tau_5(J, L, S)$. Now consider the ordering in Table 9.2: now the largest factors are $\tau_1(I, D, L, J, H)$ and $\tau_2(D, L, S, J, H)$, which are much bigger.

$$\begin{aligned}
& \sum_D \sum_C \psi_D(D, C) \sum_H \sum_L \sum_S \psi_J(J, L, S) \underbrace{\sum_I \psi_I(I) \psi_S(S, I) \sum_G \psi_G(G, I, D) \psi_L(L, G) \psi_H(H, G, J)}_{\tau_1(I, D, L, J, H)} \\
& \sum_D \sum_C \psi_D(D, C) \sum_H \sum_L \sum_S \psi_J(J, L, S) \underbrace{\sum_I \psi_I(I) \psi_S(S, I) \tau_1(I, D, L, J, H)}_{\tau_2(D, L, S, J, H)} \\
& \sum_D \sum_C \psi_D(D, C) \sum_H \sum_L \underbrace{\sum_S \psi_J(J, L, S) \tau_2(D, L, S, J, H)}_{\tau_3(D, L, J, H)} \\
& \sum_D \sum_C \psi_D(D, C) \sum_H \underbrace{\sum_L \tau_3(D, L, J, H)}_{\tau_4(D, J, H)} \\
& \sum_D \sum_C \psi_D(D, C) \underbrace{\sum_H \tau_4(D, J, H)}_{\tau_5(D, J)} \\
& \sum_D \underbrace{\sum_C \psi_D(D, C) \tau_5(D, J)}_{\tau_6(D, J)} \\
& \underbrace{\sum_D \tau_6(D, J)}_{\tau_7(J)}
\end{aligned}$$

Table 9.2: Eliminating variables from Figure 4.38 in the order G, I, S, L, H, C, D .

We can determine the size of the largest factor graphically, without worrying about the actual numerical values of the factors, by running the VE algorithm ‘‘symbolically’’. When we eliminate a variable z_t , we connect together all variables that share a factor with z_t (to reflect the new temporary factor τ'_t). The edges created by this process are called **fill-in edges**. For example, Figure 9.18 shows the fill-in edges introduced when we eliminate in the C, D, I, \dots order. The first two steps do not introduce any fill-ins, but when we eliminate I , we connect G and S , to capture the temporary factor

$$\tau'_3(G, S, I) = \psi_S(S, I) \psi_I(I) \tau_2(G, I) \quad (9.90)$$

Let \mathcal{G}_\prec be the (undirected) graph induced by applying variable elimination to \mathcal{G} using elimination ordering \prec . The temporary factors generated by VE correspond to maximal **cliques** in the graph \mathcal{G}_\prec . For example, with ordering (C, D, I, H, G, S, L) , the maximal cliques are as follows:

$$\{C, D\}, \{D, I, G\}, \{G, L, S, J\}, \{G, J, H\}, \{G, I, S\} \quad (9.91)$$

It is clear that the time complexity of VE is

$$\sum_{c \in \mathcal{C}(G_\prec)} K^{|c|} \quad (9.92)$$

where $\mathcal{C}(\mathcal{G})$ are the (maximal) cliques in graph \mathcal{G} , $|c|$ is the size of the clique c , and we assume for notational simplicity that all the variables have K states each.

Let us define the **induced width** of a graph given elimination ordering \prec , denoted w_\prec , as the size of the largest factor (i.e., the largest clique in the induced graph) minus 1. Then it is easy to see that the complexity of VE with ordering \prec is $O(K^{w_\prec+1})$. The smallest possible induced width for a graph is known as its **treewidth**. Unfortunately finding the corresponding optimal elimination order is an NP-complete problem [Yan81; ACP87]. See Section 9.5.3 for a discussion of some approximate methods for finding good elimination orders.

9.5.3 Picking a good elimination order

Many algorithms take time (or space) which is exponential in the tree width of the corresponding graph. For example, this applies to Cholesky decompositions of sparse matrices, as well as to einsum contractions (see https://github.com/dgasmith/opt_einsum). Hence we would like to find an elimination ordering that minimizes the width. We say that an ordering π is a **perfect elimination ordering** if it does not introduce any fill-in edges. Every graph that is already triangulated (e.g., a tree) has a perfect elimination ordering. We call such graphs **decomposable**.

In general, we will need to add fill-in edges to ensure the resulting graph is decomposable. Different orderings can introduce different numbers of fill-in edges, which affects the width of the resulting chordal graph; for example, compare Table 9.1 to Table 9.2.

Choosing an elimination ordering with minimal width is NP-complete [Yan81; ACP87]. It is common to use greedy approximation known as the **min-fill heuristic**, which works as follows: eliminate any node which would not result in any fill-ins (i.e., all of whose uneliminated neighbors already form a clique); if there is no such node, eliminate the node which would result in the minimum number of fill-in edges. When nodes have different weights (e.g., representing different numbers of states), we can use the **min-weight heuristic**, where we try to minimize the weight of the created cliques at each step.

Of course, many other methods are possible. See [Heg06] for a general survey. [Kja90; Kja92] compared simulated annealing with the above greedy method, and found that it sometimes works better (although it is much slower). [MJ97] approximate the discrete optimization problem by a continuous optimization problem. [BG96] present a randomized approximation algorithm. [Gil88] present the nested dissection order, which is always within $O(\log N)$ of optimal. [Ami01] discuss various constant-factor appoximation algorithms. [Dav+04] present the **AMD** or approximate minimum degree ordering algorithm, which is implemented in Matlab.⁴ The **METIS** library can be used for finding elimination orderings for large graphs; this implements the **nested dissection** algorithm [GT86]. For a planar graph with N nodes, the resulting treewidth will have the optimal size of $O(N^{3/2})$.

9.5.4 Computational complexity of exact inference

We have seen that variable elimination takes $O(NK^{w+1})$ time to compute the marginals for a graph with N nodes, and treewidth w , where each variable has K states. If the graph is densely connected, then $w = O(N)$, and so inference will take time exponential in N .

4. See the description of the symamd command at <https://bit.ly/31N6E2b>. (“sym” stands for symbolic, “amd” stands approximate minimum degree.)

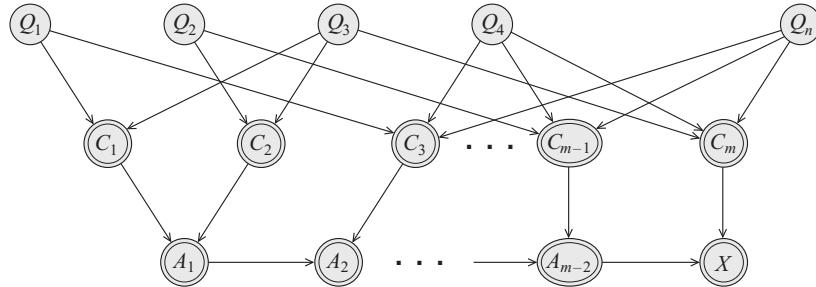


Figure 9.19: Encoding a 3-SAT problem on n variables and m clauses as a DGM. The Q_s variables are binary random variables. The C_t variables are deterministic functions of the Q_s 's, and compute the truth value of each clause. The A_t nodes are a chain of AND gates, to ensure that the CPT for the final x node has bounded size. The double rings denote nodes with deterministic CPDs. From Figure 9.1 of [KF09a]. Used with kind permission of Daphne Koller.

Of course, just because some particular algorithm is slow doesn't mean that there isn't some smarter algorithm out there. Unfortunately, this seems unlikely, since it is easy to show that exact inference for discrete graphical models is NP-hard [DL93]. The proof is a simple reduction from the satisfiability problem. In particular, note that we can encode any 3-SAT problem as a DPGM with deterministic links, as shown in Figure 9.19. We clamp the final node, x , to be on, and we arrange the CPTs so that $p(x = 1) > 0$ iff there is a satisfying assignment. Computing any posterior marginal requires evaluating the normalization constant, $p(x = 1)$, so inference in this model implicitly solves the SAT problem.

In fact, exact inference is #P-hard [Rot96], which is even harder than NP-hard. The intuitive reason for this is that to compute the normalizing constant, we have to *count* how many satisfying assignments there are. (By contrast, MAP estimation is provably easier for some model classes [GPS89], since, intuitively speaking, it only requires finding one satisfying assignment, not counting all of them.) Furthermore, even approximate inference is computationally hard in general [DL93; Rot96].

The above discussion was just concerned with inferring the states of discrete hidden variables. When we have continuous hidden variables, the problem can be even harder, since even a simple two-node graph, of the form $z \rightarrow y$, can be intractable to invert if the variables are high dimensional and do not have a conjugate relationship (Section 3.4). Inference in mixed discrete-continuous models can also be hard [LP01].

As a consequence of these hardness results, we often have to resort to approximate inference methods, such as variational inference (Chapter 10) and Monte Carlo inference (Chapter 11).

9.5.5 Drawbacks of VE

Consider using VE to compute all the marginals in a chain-structured graphical model, such as an HMM. We can easily compute the final marginal $p(z_T | \mathbf{y})$ by eliminating all the nodes z_1 to z_{T-1} in order. This is equivalent to the forwards algorithm, and takes $O(K^2 T)$ time, as we discussed in Section 9.2.3. But now suppose we want to compute $p(z_{T-1} | \mathbf{y})$. We have to run VE again, at a cost

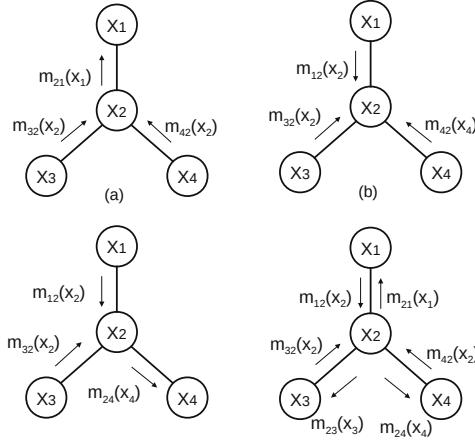


Figure 9.20: Sending multiple messages along a tree. (a) z_1 is root. (b) z_2 is root. (c) z_4 is root. (d) All of the messages needed to compute all singleton marginals. Adapted from Figure 4.3 of [Jor07].

of $O(K^2T)$ time. So the total cost to compute all the marginals is $O(K^2T^2)$. However, we know that we can solve this problem in $O(K^2T)$ using the forwards-backwards, as we discussed in Section 9.2.3. The difference is that FB caches the messages computed on the forwards pass, so it can reuse them later. (Caching previously computed results is the core idea behind dynamic programming.)

The same problem arises when applying VE to trees. For example, consider the 4-node tree in Figure 9.20. We can compute $p(z_1|\mathbf{y})$ by eliminating $z_{2:4}$; this is equivalent to sending messages up to z_1 (the messages correspond to the τ factors created by VE). Similarly we can compute $p(z_2|\mathbf{y})$, $p(z_3|\mathbf{y})$ and then $p(z_4|\mathbf{y})$. We see that some of the messages used to compute the marginal on one node can be re-used to compute the marginals on the other nodes. By storing the messages for later re-use, we can compute all the marginals in $O(K^2T)$ time, as we show in Section 9.3.

The question is: how do we get these benefits of message passing on a tree when the graph is not a tree? We give the answer in Section 9.6.

9.6 The junction tree algorithm (JTA)

The **junction tree algorithm** or **JTA** is a generalization of variable elimination that lets us efficiently compute all the posterior marginals without repeating redundant work, by using dynamic programming, thus avoiding the problems mentioned in Section 9.5.5. The basic idea is to convert the graph into a special kind of tree, known as a **junction tree** (also called a **join tree**, or **clique tree**), and then to run belief propagation (message passing) on this tree. We can create the join tree by running variable elimination “symbolically”, as discussed in Section 9.5.2, and adding the generated fill-in edges to the graph. The resulting chordal graph can then be converted to a tree, as explained in Supplementary Section 9.2.1. Once we have a tree, we can perform message passing on it, using a variant of the method Section 9.3.2. See Supplementary Section 9.2.2 for details.

9.7 Inference as optimization

In this section, we discuss how to perform posterior inference by solving an optimization problem, which is often computationally simpler. See also [Supplementary](#) Section 9.3.

9.7.1 Inference as backpropagation

In this section, we discuss how to compute posterior marginals in a graphical model using automatic differentiation. For notational simplicity, we focus on undirected graphical models, where the joint can be represented as an exponential family (Section 2.4) follows:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_c \psi_c(\mathbf{x}_c) = \exp\left(\sum_c \boldsymbol{\eta}_c^\top \mathcal{T}(\mathbf{x}_c) - \log A(\boldsymbol{\eta})\right) = \exp(\boldsymbol{\eta}^\top \mathcal{T}(\mathbf{x}) - \log A(\boldsymbol{\eta})) \quad (9.93)$$

where ψ_c is the potential function for clique c , $\boldsymbol{\eta}_c$ are the natural parameters for clique c , $\mathcal{T}(\mathbf{x}_c)$ are the corresponding sufficient statistics, and $A = \log Z$ is the log partition function.

We will consider pairwise models (with node and edge potentials), and discrete variables. The natural parameters are the node and edge log potentials, $\boldsymbol{\eta} = (\{\eta_{s;j}\}, \{\eta_{s,t;j,k}\})$, and the sufficient statistics are node and edge indicator functions, $\mathcal{T}(\mathbf{x}) = (\{\mathbb{I}(x_s = j)\}, \{\mathbb{I}(x_s = j, x_t = k)\})$. (Note: we use $s, t \in \mathcal{V}$ to index nodes and $j, k \in \mathcal{X}$ to index states.)

The mean of the sufficient statistics are given by

$$\boldsymbol{\mu} = \mathbb{E}[\mathcal{T}(\mathbf{x})] = (\{p(x_s = j)\}_s, \{p(x_s = j, x_t = k)\}_{s \neq t}) = (\{\mu_{s;j}\}_s, \{\mu_{s,t;j,k}\}_{s \neq t}) \quad (9.94)$$

The key result, from Equation (2.236), is that $\boldsymbol{\mu} = \nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta})$. Thus as long as we have a function that computes $A(\boldsymbol{\eta}) = \log Z(\boldsymbol{\eta})$, we can use automatic differentiation (Section 6.2) to compute gradients, and then we can extract the corresponding node marginals from the gradient vector. If we have evidence (known values) on some of the variables, we simply “clamp” the corresponding entries to 0 or 1 in the node potentials.

The observation that probabilistic inference can be performed using automatic differentiation has been discovered independently by several groups (e.g., [Dar03; PD03; Eis16; ASM17]). It also lends itself to the development of differentiable approximations to inference (see e.g., [MB18]).

9.7.1.1 Example: inference in a small model

As a concrete example, consider a small chain structured model $x_1 - x_2 - x_3$, where each node has K states. We can represent the node potentials as $K \times 1$ tensors (table of numbers), and the edge potentials by $K \times K$ tensors. The partition function is given by

$$Z(\boldsymbol{\psi}) = \sum_{x_1, x_2, x_3} \psi_1(x_1) \psi_2(x_2) \psi_3(x_3) \psi_{12}(x_1, x_2) \psi_{23}(x_2, x_3) \quad (9.95)$$

Let $\boldsymbol{\eta} = \log(\boldsymbol{\psi})$ be the log potentials, and $A(\boldsymbol{\eta}) = \log Z(\boldsymbol{\eta})$ be the log partition function. We can compute the single node marginals $\boldsymbol{\mu}_s = p(x_s = 1 : K)$ using $\boldsymbol{\mu}_s = \nabla_{\boldsymbol{\eta}_s} A(\boldsymbol{\eta})$, and the pairwise marginals $\boldsymbol{\mu}_{s,t}(j, k) = p(x_s = j, x_t = k)$ using $\boldsymbol{\mu}_{s,t} = \nabla_{\boldsymbol{\eta}_{s,t}} A(\boldsymbol{\eta})$.

We can compute the partition function Z efficiently use numpy’s `einsum` function, which implements tensor contraction using Einstein summation notation. We label each dimension of the tensors

by A, B, and C, so einsum knows how to match things up. We then compute gradients using an auto-diff library.⁵ The result is that inference can be done in two lines of Python code, as shown in Listing 9.1:

Listing 9.1: Computing marginals from derivative of log partition function

```
import jax.numpy as jnp
from jax import grad

logZ_fun = lambda logpots: np.log(jnp.einsum("A,B,C,AB,BC",
    *[jnp.exp(lp) for lp in logpots]))
probs = grad(logZ_fun)(logpots)
```

To perform conditional inference, such as $p(x_s = k|x_t = e)$, we multiply in one-hot indicator vectors to clamp x_t to the value e so that the unnormalized joint only assigns non-zero probability to state combinations that are valid. We then sum over all values of the unclamped variables to get the constrained partition function Z_e . The gradients will now give us the marginals conditioned on the evidence [Dar03].

9.7.2 Perturb and MAP

In this section, we discuss how to draw posterior samples from a graphical model by leveraging optimization as a subroutine. The basic idea is to make S copies of the model, each of which has slightly perturbed versions of the parameters, $\theta_s = \theta_s + \epsilon_s$, and then to compute the MAP estimate, $\mathbf{x}_s = \text{argmax} p(\mathbf{x}|\mathbf{y}; \theta_s)$. For a suitably chosen noise distribution for ϵ_s , this technique — known as **perturb-and-MAP** — can be shown that this gives exact posterior samples [PY10; PY11; PY14].

9.7.2.1 Gaussian case

We first consider the case of a Gaussian MRF. Let $\mathbf{x} \in \mathbb{R}^N$ be the vector of hidden states with prior

$$p(\mathbf{x}) \propto \mathcal{N}(\mathbf{G}\mathbf{x}|\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p) \propto \exp\left(-\frac{1}{2}\mathbf{x}^\top \mathbf{K}_x \mathbf{x} + \mathbf{h}_x^\top \mathbf{x}\right) \quad (9.96)$$

where $\mathbf{G} \in \mathbb{R}^{K \times N}$ is a matrix that represents prior dependencies (e.g., pairwise correlations), $\mathbf{K}_x = \mathbf{G}^\top \boldsymbol{\Sigma}_p^{-1} \mathbf{G}$, and $\mathbf{h}_x = \mathbf{G}^\top \boldsymbol{\Sigma}_p^{-1} \boldsymbol{\mu}_p$. Let $\mathbf{y} \in \mathbb{R}^M$ be the measurements with likelihood

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{H}\mathbf{x} + \mathbf{c}, \boldsymbol{\Sigma}_n) \propto \exp\left(-\frac{1}{2}\mathbf{x}^\top \mathbf{K}_{y|x} \mathbf{x} + \mathbf{h}_{y|x}^\top \mathbf{x} - \frac{1}{2}\mathbf{y}^\top \boldsymbol{\Sigma}_n^{-1} \mathbf{y}\right) \quad (9.97)$$

where $\mathbf{H} \in \mathbb{R}^{M \times N}$ represents dependencies between the hidden and visible variables, $\mathbf{K}_{y|x} = \mathbf{H}^\top \boldsymbol{\Sigma}_n^{-1} \mathbf{H}$ and $\mathbf{h}_{y|x} = \mathbf{H}^\top \boldsymbol{\Sigma}_n^{-1} (\mathbf{y} - \mathbf{c})$. The posterior is given by the following (cf. one step of the information filter in Section 8.2.4)

$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad (9.98)$$

$$\boldsymbol{\Sigma}^{-1} = \mathbf{K} = \mathbf{G}^\top \boldsymbol{\Sigma}_p^{-1} \mathbf{G} + \mathbf{H}^\top \boldsymbol{\Sigma}_n^{-1} \mathbf{H} \quad (9.99)$$

$$\boldsymbol{\mu} = \mathbf{K}(\mathbf{G}^\top \boldsymbol{\Sigma}_p^{-1} \boldsymbol{\mu}_p + \mathbf{H}^\top \boldsymbol{\Sigma}_n^{-1} (\mathbf{y} - \mathbf{c})) \quad (9.100)$$

5. See [ugm_inf_autodiff.py](#) for the full (JAX) code, and see <https://github.com/srush/ProbTalk> for a (PyTorch) version by Sasha Rush.

where we have assumed $\mathbf{K} = \mathbf{K}_x + \mathbf{K}_{y|x}$ is invertible (although the prior or likelihood on their own may be singular).

The K rows of $\mathbf{G} = [\mathbf{g}_1^\top; \dots; \mathbf{g}_K^\top]$ and the M rows of $\mathbf{H} = [\mathbf{h}_1^\top; \dots; \mathbf{h}_M^\top]$ can be combined into the L rows of $\mathbf{F} = [\mathbf{f}_1^\top; \dots; \mathbf{f}_L^\top]$, which define the linear constraints of the system. If we assume that Σ_p and Σ_n are diagonal, then the structure of the graphical model is uniquely determined by the sparsity of \mathbf{F} . The resulting posterior factorizes as a product of L Gaussian ‘‘experts’’:

$$p(\mathbf{x}|\mathbf{y}) \propto \prod_{l=1}^L \exp\left(-\frac{1}{2}\mathbf{x}^\top \mathbf{K}_l \mathbf{x} + \mathbf{h}_l^\top \mathbf{x}\right) \propto \prod_{l=1}^L \mathcal{N}(\mathbf{f}_l^\top \mathbf{x}; \mu_l, \Sigma_l) \quad (9.101)$$

where Σ_l equals $\Sigma_{p,l,l}$ for $l = 1 : K$ and equals $\Sigma_{n,l',l'}$ for $l = K + 1 : L$ where $l' = l - K$. Similarly $\mu_l = \mu_{p,l}$ for $l = 1 : K$ and $\mu_l = (y_{l'} - c_{l'})$ for $l = K + 1 : L$.

To apply perturb and MAP, we proceed as follows. First perturb the prior mean by sampling $\tilde{\boldsymbol{\mu}}_p \sim \mathcal{N}(\boldsymbol{\mu}_p, \Sigma_p)$, and perturb the measurements by sampling $\tilde{\mathbf{y}} \sim \mathcal{N}(\mathbf{y}, \Sigma_n)$. (Note that this is equivalent to first perturbing the linear term in each information form potential, using $\tilde{\mathbf{h}}_l = \mathbf{h}_l + \mathbf{f}_l \Sigma_l^{-\frac{1}{2}} \boldsymbol{\epsilon}_l$, where $\boldsymbol{\epsilon}_l \sim \mathcal{N}(0, 1)$.) Then compute the MAP estimate for \mathbf{x} using the perturbed parameters:

$$\tilde{\mathbf{x}} = \mathbf{K}^{-1} \mathbf{G}^\top \Sigma_p^{-1} \tilde{\boldsymbol{\mu}}_p + \mathbf{K}^{-1} \mathbf{H}^\top \Sigma_n^{-1} (\tilde{\mathbf{y}} - \mathbf{c}) \quad (9.102)$$

$$= \underbrace{\mathbf{K}^{-1} \mathbf{G}^\top \Sigma_p^{-1}}_{\mathbf{A}} (\boldsymbol{\mu}_p + \boldsymbol{\epsilon}_\mu) + \underbrace{\mathbf{K}^{-1} \mathbf{H}^\top \Sigma_n^{-1}}_{\mathbf{B}} (\mathbf{y} + \boldsymbol{\epsilon}_y - \mathbf{c}) \quad (9.103)$$

$$= \boldsymbol{\mu} + \mathbf{A} \boldsymbol{\epsilon}_\mu + \mathbf{B} \boldsymbol{\epsilon}_y \quad (9.104)$$

We see that $\mathbb{E}[\tilde{\mathbf{x}}] = \boldsymbol{\mu}$ and $\mathbb{E}[(\tilde{\mathbf{x}} - \boldsymbol{\mu})(\tilde{\mathbf{x}} - \boldsymbol{\mu})^\top] = \mathbf{K}^{-1} = \Sigma$, so the method produces exact samples.

This approach is very scalable, since compute the MAP estimate of sparse GMRFs (i.e., posterior mean) can be done efficiently using conjugate gradient solvers. Alternatively we can use loopy belief propagation (Section 9.4), which can often compute the exact posterior mean (see e.g., [WF01a; JMW06; Bic09; Du+18]).

9.7.2.2 Discrete case

In [PY11; PY14] they extend perturb-and-MAP to the case of discrete graphical models. This setup is more complicated, and requires the use of Gumbel noise, which can be sampled using $\boldsymbol{\epsilon} = -\log(-\log(u))$, where $u \sim \text{Unif}(0, 1)$. This noise should be added to all the potentials in the model, but as a simple approximation, it can just be added to the unary terms, i.e., the local evidence potentials. Let the score, or unnormalized log probability, of configuration \mathbf{x} given inputs \mathbf{c} be

$$S(\mathbf{x}; \mathbf{c}) = \log p(\mathbf{x}|\mathbf{c}) + \text{const} = \sum_i \log \phi_i(x_i) + \sum_{ij} \log \psi_{ij}(x_{i,j}) \quad (9.105)$$

where we have assumed a pairwise CRF for notational simplicity. If we perturb the local evidence potentials $\phi_i(k)$ by adding ϵ_{ik} to each entry, where k indexes the discrete latent states, we get $\tilde{S}(\mathbf{x}; \mathbf{c})$. We then compute a sample $\tilde{\mathbf{x}}$ by solving $\tilde{\mathbf{x}} = \text{argmax } \tilde{S}(\mathbf{x}; \mathbf{c})$. The advantage of this approach is that it can leverage efficient MAP solvers for discrete models, such as those discussed in Supplementary Section 9.3. This can in turn be used for parameter learning, and estimating the partition function [HJ12; Erm+13].

10 Variational inference

10.1 Introduction

In this chapter, we discuss **variational inference**, which reduces posterior inference to optimization. Note that VI is a large topic; this chapter just gives a high level overview. For more details, see e.g., [Jor+98; JJ00; Jaa01; WJ08; SQ05; TLG08; Zha+19b; Bro18].

10.1.1 The variational objective

Consider a model with unknown (latent) variables \mathbf{z} , known variables \mathbf{x} , and fixed parameters $\boldsymbol{\theta}$. (If the parameters are unknown, they can be added to \mathbf{z} , as we discuss later.) We assume the prior is $p_{\boldsymbol{\theta}}(\mathbf{z})$ and the likelihood is $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$, so the unnormalized joint is $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{z})$, and the posterior is $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})/p_{\boldsymbol{\theta}}(\mathbf{x})$. We assume that it is intractable to compute the normalization constant, $p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})d\mathbf{z}$, and hence intractable to compute the normalized posterior. We therefore seek an approximation to the posterior, which we denote by $q(\mathbf{z})$, such that we minimize the following loss:

$$q = \underset{q \in \mathcal{Q}}{\operatorname{argmin}} D_{\text{KL}}(q(\mathbf{z}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) \quad (10.1)$$

Since we are minimizing over functions (namely distributions q), this is called a **variational method**.

In practice we pick a parametric family \mathcal{Q} , where we use $\boldsymbol{\psi}$, known as the **variational parameters**, to specify which member of the family we are using. We can compute the best variational parameters (for given \mathbf{x}) as follows:

$$\boldsymbol{\psi}^* = \underset{\boldsymbol{\psi}}{\operatorname{argmin}} D_{\text{KL}}(q_{\boldsymbol{\psi}}(\mathbf{z}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) \quad (10.2)$$

$$= \underset{\boldsymbol{\psi}}{\operatorname{argmin}} \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} \left[\log q_{\boldsymbol{\psi}}(\mathbf{z}) - \log \left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})} \right) \right] \quad (10.3)$$

$$= \underset{\boldsymbol{\psi}}{\operatorname{argmin}} \underbrace{\mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} [\log q_{\boldsymbol{\psi}}(\mathbf{z}) - \log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) - \log p_{\boldsymbol{\theta}}(\mathbf{z})]}_{\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}|\mathbf{x})} + \log p_{\boldsymbol{\theta}}(\mathbf{x}) \quad (10.4)$$

The final term $\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log(\int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})d\mathbf{z})$ is generally intractable to compute. Fortunately, it is independent of $\boldsymbol{\psi}$, so we can drop it. This leaves us with the first term, which we write as follows:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}|\mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} [-\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) + \log q_{\boldsymbol{\psi}}(\mathbf{z})] \quad (10.5)$$

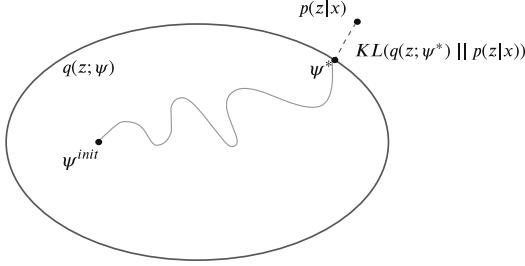


Figure 10.1: Illustration of variational inference. The large oval represents the set of variational distributions $\mathcal{Q} = \{q_\psi(z) : \psi \in \Theta\}$, where Θ is the set of possible variational parameters. The true distribution is the point $p(z|x)$, which we assume lies outside the set. Our goal is to find the best approximation to p within our variational family; this is the point ψ^* which is closest in KL divergence. We find this point by starting an optimization procedure from the random initial point ψ^{init} . Adapted from a figure by David Blei.

Minimizing this objective will minimize the KL divergence, causing our approximation to approach the true posterior. See Figure 10.1 for an illustration. In the sections below, we give two different interpretations of this objective function.

10.1.1.1 The view from physics: minimize the variational free energy

If we define $\mathcal{E}_\theta(z) = -\log p_\theta(z, x)$ as the energy, then we can rewrite the loss in Equation (10.5)

$$\mathcal{L}(\theta, \psi|x) = \mathbb{E}_{q_\psi(z)} [\mathcal{E}_\theta(z)] - \mathbb{H}(q_\psi) = \text{expected energy} - \text{entropy} \quad (10.6)$$

In physics, this is known as the **variational free energy** (VFE). This is an upper bound on the **free energy** (FE), $-\log p_\theta(x)$, which follows from the fact that

$$D_{KL}(q_\psi(z) \| p_\theta(z|x)) = \mathcal{L}(\theta, \psi|x) + \log p_\theta(x) \geq 0 \quad (10.7)$$

$$\underbrace{\mathcal{L}(\theta, \psi|x)}_{\text{VFE}} \geq \underbrace{-\log p_\theta(x)}_{\text{FE}} \quad (10.8)$$

Variational inference is equivalent to minimizing the VFE. If we reach the minimum value of $-\log p_\theta(x)$, then the KL divergence term will be 0, so our approximate posterior will be exact.

10.1.1.2 The view from statistics: maximize the evidence lower bound (ELBO)

The negative of the VFE is known as the **evidence lower bound** or **ELBO** function [BKM16]:

$$\mathcal{L}(\theta, \psi|x) \triangleq \mathbb{E}_{q_\psi(z)} [\log p_\theta(x, z) - \log q_\psi(z)] = \text{ELBO} \quad (10.9)$$

The name ‘‘ELBO’’ arises because

$$\mathcal{L}(\theta, \psi|x) \leq \log p_\theta(x) \quad (10.10)$$

where $\log p_\theta(x)$ is called the ‘‘evidence’’. The inequality follows from Equation (10.8). Therefore maximizing the ELBO wrt ψ will minimize the original KL, since $\log p_\theta(x)$ is a constant wrt ψ .

(Note: we use the symbol \mathbb{L} for the ELBO, rather than \mathcal{L} , since we want to maximize \mathbb{L} but minimize \mathcal{L} .)

We can rewrite the ELBO as follows:

$$\mathbb{L}(\boldsymbol{\theta}, \boldsymbol{\psi} | \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] + \mathbb{H}(q_{\boldsymbol{\psi}}(\mathbf{z})) \quad (10.11)$$

We can interpret this

$$\text{ELBO} = \text{expected log joint} + \text{entropy} \quad (10.12)$$

The second term encourages the posterior to be maximum entropy, while the first term encourages it to be a joint MAP configuration.

We can also rewrite the ELBO as

$$\mathbb{L}(\boldsymbol{\psi} | \boldsymbol{\theta}, \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} [\log p_{\boldsymbol{\theta}}(\mathbf{x} | \mathbf{z}) + \log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\psi}}(\mathbf{z})] \quad (10.13)$$

$$= \mathbb{E}_{q_{\boldsymbol{\psi}}(\mathbf{z})} [\log p_{\boldsymbol{\theta}}(\mathbf{x} | \mathbf{z})] - D_{\text{KL}}(q_{\boldsymbol{\psi}}(\mathbf{z}) \| p_{\boldsymbol{\theta}}(\mathbf{z})) \quad (10.14)$$

We can interpret this as follows:

$$\text{ELBO} = \text{expected log likelihood} - \text{KL from posterior to prior} \quad (10.15)$$

The KL term acts like a regularizer, preventing the posterior from diverging too much from the prior.

10.1.2 Form of the variational posterior

There are two main approaches for choosing the form of the variational posterior, $q_{\boldsymbol{\psi}}(\mathbf{z} | \mathbf{x})$. In the first approach, we pick a convenient functional form, such as multivariate Gaussian, and then optimize the ELBO using gradient-based methods. This is called **fixed-form VI**, and is discussed in Section 10.2. An alternative is to make the **mean field** assumption, namely that the posterior factorizes:

$$q_{\boldsymbol{\psi}}(\mathbf{z}) = \prod_{j=1}^J q_j(\mathbf{z}_j) \quad (10.16)$$

where $q_j(\mathbf{z}_j) = q_{\boldsymbol{\psi}_j}(\mathbf{z}_j)$ is the posterior over the j 'th group of variables. We don't need to specify the functional form for each q_j . Instead, the optimal distributional form can be derived by maximizing the ELBO wrt each group of variational parameters one at a time, in a coordinate ascent manner. This is therefore called **free-form VI**, and is discussed in Section 10.3.

We now give a simple example of variational inference applied to a 2d latent vector \mathbf{z} , representing the mean of a Gaussian. The prior is $\mathcal{N}(\mathbf{z} | \bar{\mathbf{m}}, \bar{\mathbf{V}})$, and the likelihood is

$$p(\mathcal{D} | \mathbf{z}) = \prod_{n=1}^N \mathcal{N}(\mathbf{x}_n | \mathbf{z}, \Sigma) \propto \mathcal{N}(\bar{\mathbf{x}} | \mathbf{z}, \frac{1}{N} \Sigma) \quad (10.17)$$

The exact posterior, $p(\mathbf{z} | \mathcal{D}) = \mathcal{N}(\mathbf{z} | \hat{\mathbf{m}}, \hat{\mathbf{V}})$, can be computed analytically, as discussed in Section 3.4.4.1. In Figure 10.2, we compare three Gaussian variational approximations to the posterior. If q uses a full covariance matrix, it matches the exact posterior; however, this is intractable in high

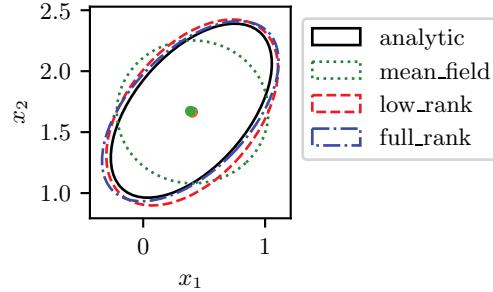


Figure 10.2: Variational approximation to the exact (Gaussian) posterior for the mean of a 2d Gaussian likelihood with a Gaussian prior. We show 3 Gaussian approximations to the posterior, using a full covariance (blue), a diagonal covariance (green), and a diagonal plus rank one covariance (red). Generated by [gaussian_2d_vi.ipynb](#).

dimensions. If q uses a diagonal covariance matrix (corresponding to the mean field approximation), we see that the approximation is over confident, which is a well-known flaw of variational inference, due to the mode-seeking nature of minimizing $D_{\text{KL}}(q \parallel p)$ (see Section 5.1.4.3 for details). Finally, if q uses a rank-1 plus diagonal approximation, we get a much better approximation; furthermore, this can be computed quite efficiently, as we discuss in Section 10.2.1.3.

10.1.3 Parameter estimation using variational EM

So far, we have assumed the model parameters $\boldsymbol{\theta}$ are known. However, we can try to estimate them by maximizing the log marginal likelihood of the dataset, $\mathcal{D} = \{\mathbf{x}_n : n = 1 : N\}$,

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{n=1}^N \log p(\mathbf{x}_n|\boldsymbol{\theta}) \quad (10.18)$$

In general, this is intractable to compute, but we discuss approximations below.

10.1.3.1 MLE for latent variable models

Suppose we have a latent variable model of the form

$$p(\mathcal{D}, \mathbf{z}_{1:N}|\boldsymbol{\theta}) = \prod_{n=1}^N p(\mathbf{z}_n|\boldsymbol{\theta})p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}) \quad (10.19)$$

as shown in Figure 10.3a. Furthermore, suppose we want to compute the MLE for $\boldsymbol{\theta}$ given the dataset $\mathcal{D} = \{\mathbf{x}_n : n = 1 : N\}$. Since the local latent variables \mathbf{z}_n are hidden, we must marginalize them out to get the local (per example) log marginal likelihood:

$$\log p(\mathbf{x}_n|\boldsymbol{\theta}) = \log \left[\int p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta})p(\mathbf{z}_n|\boldsymbol{\theta})d\mathbf{z}_n \right] \quad (10.20)$$

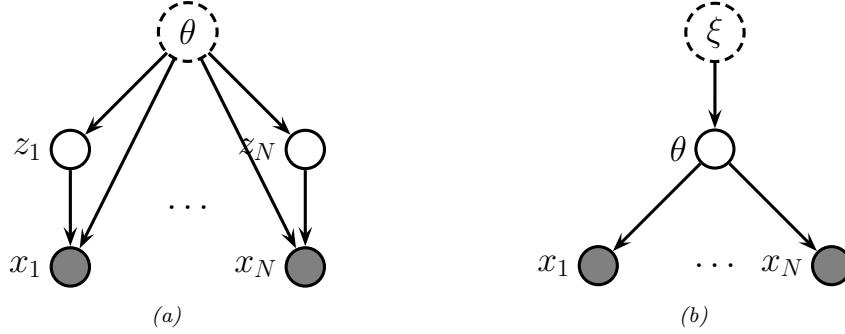


Figure 10.3: Graphical models with: (a) Local stochastic latent variables z_n and global deterministic latent parameter θ . (b) Global stochastic latent parameter θ and global deterministic latent hyper-parameter ξ . The observed variables x_n are shown by shaded circles.

Unfortunately, computing this integral is usually intractable, since it corresponds to the normalization constant of the exact posterior. Fortunately, the ELBO is a lower bound on this:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}_n | \mathbf{x}_n) \leq \log p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (10.21)$$

We can thus optimize the model parameters by maximizing

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}_{1:N} | \mathcal{D}) \triangleq \sum_{n=1}^N \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}_n | \mathbf{x}_n) \leq \log p(\mathcal{D} | \boldsymbol{\theta}) \quad (10.22)$$

This is the basis of the **variational EM** algorithm. We discuss this in more detail in Section 6.5.6.1, but the basic idea is to alternate between maximizing the ELBO wrt the variational parameters $\{\boldsymbol{\psi}_n\}$ in the E step, to give us $q_{\boldsymbol{\psi}_n}(\boldsymbol{z}_n)$, and then maximizing the ELBO (using the new $\boldsymbol{\psi}_n$) wrt the model parameters $\boldsymbol{\theta}$ in the M step. (We can also use SGD and amortized inference to speed this up, as we explain in Sections 10.1.4 to 10.1.5.)

10.1.3.2 Empirical Bayes for fully observed models

Suppose we have a fully observed model (with no local latent variables) of the form

$$p(\mathcal{D}, \boldsymbol{\theta} | \boldsymbol{\xi}) = p(\boldsymbol{\theta} | \boldsymbol{\xi}) \prod_{n=1}^N p(\mathbf{x}_n | \boldsymbol{\theta}) \quad (10.23)$$

as shown in Figure 10.3b. In the context of Bayesian parameter inference, our goal is to compute the parameter posterior:

$$p(\boldsymbol{\theta} | \mathcal{D}, \boldsymbol{\xi}) = \frac{p(\mathcal{D} | \boldsymbol{\theta}) p(\boldsymbol{\theta} | \boldsymbol{\xi})}{p(\mathcal{D} | \boldsymbol{\xi})} \quad (10.24)$$

where $\boldsymbol{\theta}$ are the global unknown model parameters (latent variables), and $\boldsymbol{\xi}$ are the hyper-parameters for the prior. If the hyper-parameters are unknown, we can estimate them using empirical Bayes (see

Section 3.7) by computing

$$\hat{\xi} = \underset{\xi}{\operatorname{argmax}} \log p(\mathcal{D}|\xi) \quad (10.25)$$

We can use variational EM to compute this, similar to Section 10.1.3.1, except now the parameters to be estimated are ξ , the latent variables are the shared global parameters θ , and the observations are the entire dataset, \mathcal{D} . We then get the lower bound

$$\log p(\mathcal{D}|\xi) \geq \mathbb{L}(\xi, \psi|\mathcal{D}) = \mathbb{E}_{q_\psi(\theta)} \left[\sum_{n=1}^N \log p(\mathbf{x}_n|\theta) \right] - D_{\text{KL}}(q_\psi(\theta) \parallel p(\theta|\xi)) \quad (10.26)$$

We optimize this wrt the parameters of the variational posterior, ψ , and wrt the prior hyper-parameters ξ .

If the prior ξ is fixed, we just need to optimize the variational parameters ψ to compute the posterior, $q_\psi(\theta|\mathcal{D})$. This is known as **variational Bayes**. See Section 10.3.3 for more details.

10.1.4 Stochastic VI

In Section 10.1.3, we saw that parameter estimation requires optimizing the ELBO for the entire dataset, which is defined as the sum of the ELBOs for each of the N data samples \mathbf{x}_n . Computing this objective can be slow if N is large. Fortunately, we can replace this objective with a stochastic approximation, which is faster to compute, and provides an unbiased estimate. In particular, at each step, we can draw a random minibatch of $B = |\mathcal{B}|$ examples from the dataset, and then make the approximation

$$\mathbb{L}(\theta, \psi_{1:N}|\mathcal{D}) = \sum_{n=1}^N \mathbb{L}(\theta, \psi_n|\mathbf{x}_n) \approx \frac{N}{B} \sum_{\mathbf{x}_n \in \mathcal{B}} \left[\mathbb{E}_{q_{\psi_n}(\mathbf{z}_n)} [\log p_\theta(\mathbf{x}_n|\mathbf{z}_n) + \log p_\theta(\mathbf{z}_n) - \log q_{\psi_n}(\mathbf{z}_n)] \right] \quad (10.27)$$

This can be used inside of a stochastic optimization algorithm, such as SGD. This is called **stochastic variational inference** or **SVI** [Hof+13], and allows VI to scale to large datasets.

10.1.5 Amortized VI

In Section 10.1.4, we saw that in each iteration of SVI, we need to optimize the local variational parameters ψ_n for each example n in the minibatch. This nested optimization can be quite slow.

An alternative approach is to train a model, known as an **inference network** or , to predict ψ_n from the observed data, \mathbf{x}_n , using $\psi_n = f_\phi^{\text{inf}}(\mathbf{x}_n)$. This technique is known as **amortized variational inference** [GG14], or **inference compilation** [LBW17], since we are reducing the cost of per-example time inference by training a model that is shared across all examples. (See also [Amo22] for a general discussion of amortized optimization.) For brevity, we will write the amortized posterior as

$$q(\mathbf{z}_n|\psi_n) = q(\mathbf{z}_n|f_\phi^{\text{inf}}(\mathbf{x}_n)) = q_\phi(\mathbf{z}_n|\mathbf{x}_n) \quad (10.28)$$

The corresponding ELBO becomes

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathcal{D}) = \sum_{n=1}^N [\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n | \mathbf{x}_n)} [\log p_{\boldsymbol{\theta}}(\mathbf{x}_n, \mathbf{z}_n) - \log q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x}_n)]] \quad (10.29)$$

If we combine this with SVI we get an amortized version of Equation (10.27). For example, if we use a minibatch of size 1, we get

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}_n) \approx N [\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}_n | \mathbf{x}_n)} [\log p_{\boldsymbol{\theta}}(\mathbf{x}_n, \mathbf{z}_n) - \log q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x}_n)]] \quad (10.30)$$

We can optimize this as shown in Algorithm 10.1. Note that the (partial) maximization wrt $\boldsymbol{\theta}$ in the M step is usually done with a gradient update, but the maximization wrt $\boldsymbol{\phi}$ in the E step is trickier, since the loss uses $\boldsymbol{\phi}$ to define an expectation operator, so we can't necessarily push the gradient operator inside; we discuss ways to optimize the variational parameters in Section 10.2 and Section 10.3.

Algorithm 10.1: Amortized stochastic variational EM

```

1 Initialize  $\boldsymbol{\theta}$ ,  $\boldsymbol{\phi}$ 
2 repeat
3   Sample  $\mathbf{x}_n \sim p_{\mathcal{D}}$ 
4   E step:  $\boldsymbol{\phi} = \text{argmax}_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}_n)$ 
5   M step:  $\boldsymbol{\theta} = \text{argmax}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}_n)$ 
6 until converged

```

10.1.6 Semi-amortized inference

Amortized SVI is widely used for fitting LVMs, e.g., for VAEs (see Section 21.2), for topic models [SS17a], for probabilistic programming [RHG16], for CRFs [TG18], etc. However, the use of an inference network can result in a suboptimal setting of the local variational parameters $\boldsymbol{\psi}_n$. This is called the **amortization gap** [CLD18]. We can close this gap by using the inference network to warm-start an optimizer for $\boldsymbol{\psi}_n$; this is known as **semi-amortized VI** [Kim+18c]. (See also [MYM18], who propose a closely related method called **iterative amortized inference**.)

An alternative approach is to use the inference network as a proposal distribution. If we combine this with importance sampling, we get the IWAE bound of Section 10.5.1. If we use this with Metropolis-Hastings, we get a VI-MCMC hybrid (see Section 10.4.5).

10.2 Gradient-based VI

In this section, we will choose some convenient form for $q_{\boldsymbol{\psi}}(\mathbf{z})$, such as a Gaussian for continuous \mathbf{z} , or a product of categoricals for discrete \mathbf{z} , and then optimize the ELBO using gradient based methods.

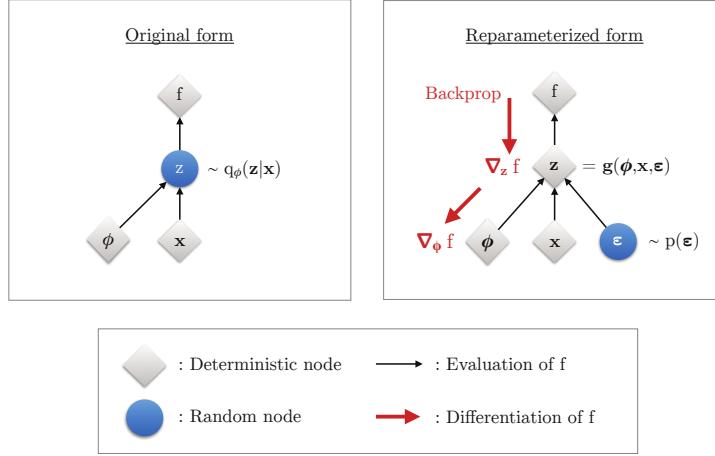


Figure 10.4: Illustration of the reparameterization trick. The objective f depends on the variational parameters ϕ , the observed data x , and the latent random variable $z \sim q_\phi(z|x)$. On the left, we show the standard form of the computation graph. On the right, we show a reparameterized form, in which we move the stochasticity into the noise source ϵ , and compute z deterministically, $z = g(\phi, x, \epsilon)$. The rest of the graph is deterministic, so we can backpropagate the gradient of the scalar f wrt ϕ through z and into ϕ . From Figure 2.3 of [KW19a]. Used with kind permission of Durk Kingma.

The gradient wrt the generative parameters θ is easy to compute, since we can push gradients inside the expectation, and use a single Monte Carlo sample:

$$\nabla_\theta L(\theta, \phi | x) = \nabla_\theta \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x, z) - \log q_\phi(z|x)] \quad (10.31)$$

$$= \mathbb{E}_{q_\phi(z|x)} [\nabla_\theta \{\log p_\theta(x, z) - \log q_\phi(z|x)\}] \quad (10.32)$$

$$\approx \nabla_\theta \log p_\theta(x, z^s) \quad (10.33)$$

where $z^s \sim q_\phi(z|x)$. This is an unbiased estimate of the gradient, so can be used with SGD.

The gradient wrt the inference parameters ϕ is harder to compute since

$$\nabla_\phi L(\theta, \phi | x) = \nabla_\phi \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x, z) - \log q_\phi(z|x)] \quad (10.34)$$

$$\neq \mathbb{E}_{q_\phi(z|x)} [\nabla_\phi \{\log p_\theta(x, z) - \log q_\phi(z|x)\}] \quad (10.35)$$

However, we can often use the reparameterization trick, which we discuss in Section 10.2.1. If not, we can use blackbox VI, which we discuss in Section 10.2.3.

10.2.1 Reparameterized VI

In this section, we discuss the **reparameterization trick** for taking gradients wrt distributions over continuous latent variables $z \sim q_\phi(z|x)$. We explain this in detail in Section 6.3.5, but we summarize the basic idea here.

The key trick is to rewrite the random variable $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ as some differentiable (and invertible) transformation g of another random variable $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$, which does not depend on ϕ , i.e., we assume we can write

$$\mathbf{z} = g(\phi, \mathbf{x}, \boldsymbol{\epsilon}) \quad (10.36)$$

For example,

$$\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma})) \iff \mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\epsilon} \odot \boldsymbol{\sigma}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (10.37)$$

Using this, we have

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(\mathbf{z})] \quad \text{s.t. } \mathbf{z} = g(\phi, \mathbf{x}, \boldsymbol{\epsilon}) \quad (10.38)$$

where we define

$$f_{\theta, \phi}(\mathbf{z}) = \log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}) \quad (10.39)$$

Hence

$$\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] = \nabla_\phi \mathbb{E}_{p(\boldsymbol{\epsilon})}[f(\mathbf{z})] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla_\phi f(\mathbf{z})] \quad (10.40)$$

which we can approximate with a single Monte Carlo sample. This lets us propagate gradients back through the f function. See Figure 10.4 for an illustration. This is called **reparameterized VI** or **RVI**.

Since we are now working with the random variable $\boldsymbol{\epsilon}$, we need to use the change of variables formula to compute

$$\log q_\phi(\mathbf{z}|\mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log \left| \det \left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right) \right| \quad (10.41)$$

where $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}$ is the Jacobian:

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \dots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \dots & \frac{\partial z_k}{\partial \epsilon_k} \end{pmatrix} \quad (10.42)$$

We design the transformation $\mathbf{z} = g(\boldsymbol{\epsilon})$ such that this Jacobian is tractable to compute. We give some examples below.

10.2.1.1 Gaussian with diagonal covariance (mean field)

Suppose we use a fully factorized Gaussian posterior. Then the reparameterization process becomes

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (10.43)$$

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon} \quad (10.44)$$

where the inference network generates the parameters of the transformation:

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = f_{\phi}^{\text{inf}}(\mathbf{x}) \quad (10.45)$$

Thus to sample from the posterior $q_{\phi}(\mathbf{z}|\mathbf{x})$, we sample $\epsilon \mathcal{N}(\mathbf{0}, \mathbf{I})$, and then compute \mathbf{z} .

Given the sample, we need to evaluate the ELBO:

$$f(\mathbf{z}) = \log p_{\theta}(\mathbf{x}|\mathbf{z}) + \log p_{\theta}(\mathbf{z}) - \log q_{\phi}(\mathbf{z}|\mathbf{x}) \quad (10.46)$$

To evaluate the $p_{\theta}(\mathbf{x}|\mathbf{z})$ term, we can just plug \mathbf{z} into the likelihood. To evaluate the $\log q_{\phi}(\mathbf{z}|\mathbf{x})$ term, we need to use the change of variables formula from Equation (10.41). The Jacobian is given by $\frac{\partial \mathbf{z}}{\partial \epsilon} = \text{diag}(\boldsymbol{\sigma})$. Hence

$$\log q_{\phi}(\mathbf{z}|\mathbf{x}) = \sum_{k=1}^K [\log \mathcal{N}(\epsilon_k|0, 1) - \log \sigma_k] = - \sum_{k=1}^K \left[\frac{1}{2} \log(2\pi) + \frac{1}{2} \epsilon_k^2 + \log \sigma_k \right] \quad (10.47)$$

Finally, to evaluate the $p(\mathbf{z})$ term, we can use the transformation $\mathbf{z} = \mathbf{0} + \mathbf{1} \odot \boldsymbol{\epsilon}$, so the Jacobian is the identity and we get

$$\log p(\mathbf{z}) = \sum_{k=1}^K \left[\frac{1}{2} z_k^2 + \frac{1}{2} \log(2\pi) \right] \quad (10.48)$$

An alternative is to use the objective

$$f'(\mathbf{z}) = \log p_{\theta}(\mathbf{x}|\mathbf{z}) + D_{\text{KL}}(q_{\phi}(\mathbf{Z}|\mathbf{x}) \parallel p_{\theta}(\mathbf{Z})) \quad (10.49)$$

In some cases, we evaluate the second term analytically, without needing Monte Carlo. For example, if we assume a diagonal Gaussian prior, $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$, and diagonal gaussian posterior, $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}))$, we can use Equation (5.78) to compute the KL in closed form:

$$D_{\text{KL}}(q \parallel p) = -\frac{1}{2} \sum_{k=1}^K [\log \sigma_k^2 - \sigma_k^2 - \mu_k^2 + 1] \quad (10.50)$$

The objective $f'(\mathbf{z})$ is often lower variance than $f(\mathbf{z})$, since it computes the KL analytically. However, it is harder to generalize this objective to settings where the prior and/or posterior are not Gaussian.

10.2.1.2 Gaussian with full covariance

Now consider using a full covariance Gaussian posterior. We will compute a Cholesky decomposition of the covariance, $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T$, where \mathbf{L} is a lower triangular matrix with non-zero entries on the diagonal. Hence the reparameterization becomes

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (10.51)$$

$$\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon} \quad (10.52)$$

The Jacobian of this affine transformation is $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \mathbf{L}$. Since \mathbf{L} is a triangular matrix, its determinant is the product of its main diagonal, so

$$\log \left| \det \frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right| = \sum_{k=1}^K \log |L_{kk}| \quad (10.53)$$

We can compute \mathbf{L} using

$$\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma}) \quad (10.54)$$

where \mathbf{M} is a masking matrix with 0s on and above the diagonal, and 1s below the diagonal, and where $(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}')$ is predicted by the inference network. With this construction, the diagonal entries of \mathbf{L} are given by $\boldsymbol{\sigma}$, so

$$\log \left| \det \frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right| = \sum_{k=1}^K \log |L_{kk}| = \sum_{k=1}^K \log \sigma_k \quad (10.55)$$

10.2.1.3 Gaussian with low-rank plus diagonal covariance

In high dimensions, an efficient alternative to using a Cholesky decomposition is the factor decomposition

$$\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^\top + \mathbf{C}^2 \quad (10.56)$$

where \mathbf{B} is the factor loading matrix of size $d \times f$, where $f \ll d$ is the number of factors, d is the dimensionality of \mathbf{z} , and $\mathbf{C} = \text{diag}(c_1, \dots, c_d)$. This reduces the total number of variational parameters from $d + d(d+1)/2$ to $(f+2)d$. In [ONS18], they called this approach **VAF**C (short for variational approximation with factor covariance).

In the special case where $f = 1$, the covariance matrix becomes

$$\boldsymbol{\Sigma} = \mathbf{b}\mathbf{b}^\top + \text{diag}(\mathbf{c}^2) \quad (10.57)$$

In this case, it is possible to compute the natural gradient (Section 6.4) of the ELBO in closed form in $O(d)$ time, as shown in [Tra+20b; TND21], who call the approach **NAGVAC-1** (natural gradient Gaussian variational approximation). This can result in much faster convergence than following the normal gradient.

In Section 10.1.2, we show that this low rank approximation is much better than a diagonal approximation. See [Supplementary](#) Section 10.1 for more examples.

10.2.1.4 Other variational posteriors

Many other kinds of distribution can be written in a reparameterizable way, as described in [Moh+20]. This includes standard exponential family distributions, such as the gamma and Dirichlet, as well as more exotic forms, such as inverse autoregressive flows (see Section 10.4.3).

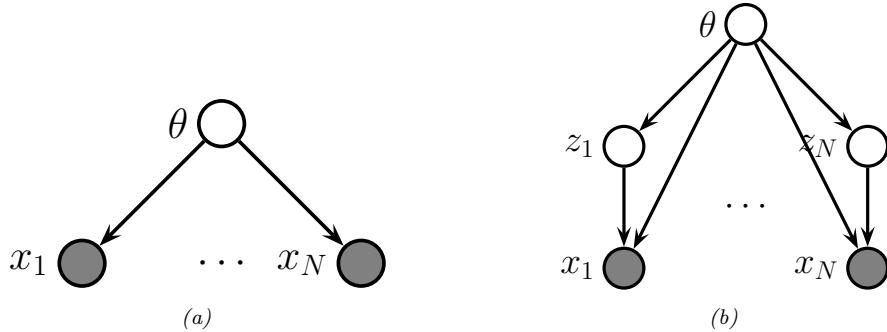


Figure 10.5: Graphical models with (a) Global latent parameter θ and observed variables $x_{1:N}$. (b) Local latent variables $z_{1:N}$, global latent parameter θ , and observed variables $x_{1:N}$.

10.2.1.5 Example: Bayesian parameter inference

In this section, we use reparameterized SVI to infer the posterior for the parameters of a Gaussian mixture model (GMM). We will marginalize out the discrete latent variables, so just need to approximate the posterior over the global latent, $p(\theta|\mathcal{D})$. This is sometimes called a “**collapsed**” model, since we have marginalized out all the local latent variables. That is, we have converted the model in Figure 10.5b to the one in Figure 10.5a. We choose a factored (mean field) variational posterior that is conjugate to the likelihood, but is also reparameterizable. This lets us fit the posterior with SGD.

For simplicity, we assume diagonal covariance matrices for each Gaussian mixture component. Thus the likelihood for one datapoint, $\mathbf{x} \in \mathbb{R}^D$, is

$$p(\mathbf{x}|\boldsymbol{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \text{diag}(\boldsymbol{\lambda}_k)^{-1}) \quad (10.58)$$

where $\boldsymbol{\mu}_k = (\mu_{k1}, \dots, \mu_{kD})$ are the means, $\boldsymbol{\lambda}_k = (\lambda_{k1}, \dots, \lambda_{kD})$ are the precisions, and $\boldsymbol{\pi} = (\pi_1, \dots, \pi_K)$ are the mixing weights. We use the following prior for these parameters:

$$p_{\xi}(\boldsymbol{\theta}) = \left[\prod_{k=1}^K \prod_{d=1}^D \mathcal{N}(\mu_{kd}|0, 1) \text{Ga}(\lambda_{kd}|5, 5) \right] \text{Dir}(\boldsymbol{\pi}|\mathbf{1}) \quad (10.59)$$

where ξ are the hyperparameters. We assume the following mean field posterior:

$$q_{\psi}(\boldsymbol{\theta}) = \left[\prod_{k=1}^K \prod_{d=1}^D \mathcal{N}(\mu_{kd}|m_{kd}, s_{kd}) \text{Ga}(\lambda_{kd}|\alpha_{kd}, \beta_{kd}) \right] \text{Dir}(\boldsymbol{\pi}|\boldsymbol{c}) \quad (10.60)$$

where $\psi = (\mathbf{m}_{1:K,1:D}, \mathbf{s}_{1:K,1:D}, \boldsymbol{\alpha}_{1:K,1:D}, \boldsymbol{\beta}_{1:K,1:D}, \mathbf{c})$ are the variational parameters for θ .

We can compute the ELBO using

$$\mathcal{L}(\xi, \psi | \mathcal{D}) = \mathbb{E}_{q_\psi(\theta)} [\log p(\mathcal{D}|\theta) + \log p_\xi(\theta) - \log q_\psi(\theta)] \quad (10.61)$$

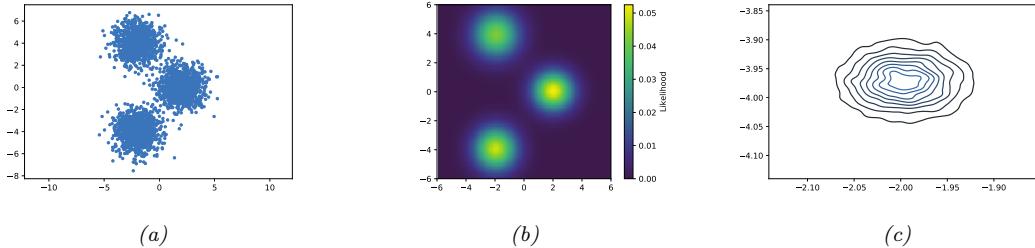


Figure 10.6: SVI for fitting a mixture of 3 Gaussians in 2d. (a) 3000 training points. (b) Fitted density, plugging in the posterior mean parameters. (c) Kernel density estimate fit to 10,000 samples from $q(\mu_1|\psi)$. Generated by [svi_gmm_demo_2d.ipynb](#).

Since the distributions are reparameterizable, we can and push gradients inside this expression. We can approximate the expectation by drawing a single posterior sample, and can approximate the log likelihood using minibatching. We can then update the variational parameters, (and optionally the hyperparameters of the prior, as we discussed in Section 10.1.3.2) using the pseudocode in Algorithm 10.2.

Algorithm 10.2: Reparameterized SVI for Bayesian parameter inference

```

1 Initialize  $\psi, \xi$ 
2 repeat
3   Sample minibatch  $\mathcal{B} = \{\mathbf{x}_b \sim \mathcal{D} : b = 1 : B\}$ 
4   Sample  $\epsilon \sim q_0$ 
5   Compute  $\tilde{\theta} = g(\psi, \epsilon)$ 
6   Compute  $\mathcal{L}(\psi|\mathcal{D}, \tilde{\theta}) = -\frac{N}{B} \sum_{\mathbf{x}_n \in \mathcal{B}} \log p(\mathbf{x}_n|\tilde{\theta}) - \log p_\xi(\tilde{\theta}) + \log q_\psi(\tilde{\theta})$ 
7   Update  $\xi := \xi - \eta \nabla_\xi \mathcal{L}(\xi, \psi|\mathcal{D}, \tilde{\theta})$ 
8   Update  $\psi := \psi - \eta \nabla_\psi \mathcal{L}(\xi, \psi|\mathcal{D}, \tilde{\theta})$ 
9 until converged

```

Figure 10.6 gives an example of this in practice. We generate a dataset from a mixture of 3 Gaussians in 2d, using $\mu_1^* = [2, 0]$, $\mu_2^* = [-2, -4]$, $\mu_3^* = [-2, 4]$, precisions $\lambda_{dk}^* = 1$, and uniform mixing weights, $\pi^* = [1/3, 1/3, 1/3]$. Figure 10.6a shows the training set of 3000 points. We fit this using SVI, with a batch size of 500, for 1000 epochs, using the Adam optimizer. Figure 10.6b shows the predictions of the fitted model. More precisely, it shows $p(\mathbf{x}|\bar{\theta})$, where $\bar{\theta} = \mathbb{E}_{q(\theta|\psi)}[\theta]$. Figure 10.6c shows a kernel density estimate fit to 10,000 samples from $q(\mu_1|\psi)$. We see that the posterior mean is $\mathbb{E}[\mu_1] \approx [-2, -4]$. Due to label switching unidentifiability, we see this matches μ_2^* rather than μ_1^* .

10.2.1.6 Example: MLE for LVMs

In this section, we consider reparameterized SVI for computing the MLE for latent variable models (LVMs) with continuous latents, such as variational autoencoders (Section 21.2). Unlike Section 10.2.1.5, we cannot analytically marginalize out the local latents. Instead we will use amortized inference, as in Section 10.1.5, which means we learn an inference network (with parameters ϕ) to predict the local variational parameters ψ_n given input x_n . If we sample a single example x_n from the dataset at each iteration, and a single latent variable z_n from the variational posterior, then we get the pseudocode in Algorithm 10.3.

Algorithm 10.3: Reparameterized amortized SVI for MLE of an LVM

```

1 Initialize  $\theta, \phi$ 
2 repeat
3   Sample  $x_n \sim p_{\mathcal{D}}$ 
4   Sample  $\epsilon_n \sim q_0$ 
5   Compute  $z_n = g(\phi, x_n, \epsilon_n)$ 
6   Compute  $\mathcal{L}(\theta, \phi | x_n, z_n) = -\log p_{\theta}(x_n, z_n) + \log q_{\phi}(z_n | x_n)$ 
7   Update  $\theta := \theta - \eta \nabla_{\theta} \mathcal{L}(\phi, \theta | x_n, z_n)$ 
8   Update  $\phi := \phi - \eta \nabla_{\phi} \mathcal{L}(\phi, \theta | x_n, z_n)$ 
9 until converged

```

10.2.2 Automatic differentiation VI

To apply Gaussian VI, we need to transform constrained parameters (such as variance terms) to unconstrained form, so they live in \mathbb{R}^D . This technique can be used for any distribution for which we can define a bijection to \mathbb{R}^D . This approach is called **automatic differentiation variational inference** or ADVI [Kuc+16]. We give the details below.

10.2.2.1 Basic idea

Our goal is to approximate the posterior $p(\theta | \mathcal{D}) \propto p(\theta)p(\mathcal{D} | \theta)$, where $\theta \in \Theta$ lives in some D -dimensional constrained parameter space. Let $T : \Theta \rightarrow \mathbb{R}^D$ be a bijective mapping that maps from the constrained space Θ to the unconstrained space \mathbb{R}^D , with inverse $T^{-1} : \mathbb{R}^D \rightarrow \Theta$. Let $\mathbf{u} = T(\theta)$ be the unconstrained latent variables. We will use a Gaussian variational approximation to the posterior for \mathbf{u} , i.e.,: $q_{\psi}(\mathbf{u}) = \mathcal{N}(\mathbf{u} | \mu_d, \Sigma)$, where $\psi = (\mu, \Sigma)$.

By the change of variable formula Equation (2.257), we have

$$p(\mathbf{u}) = p(T^{-1}(\mathbf{u})) |\det(\mathbf{J}_{T^{-1}}(\mathbf{u}))| \quad (10.62)$$

where $\mathbf{J}_{T^{-1}}$ is the Jacobian of the inverse mapping $\mathbf{u} \rightarrow \theta$. Hence the ELBO becomes

$$\mathcal{L}(\psi) = E_{\mathbf{u} \sim q_{\psi}(\mathbf{u})} [\log p(\mathcal{D} | T^{-1}(\mathbf{u})) + \log p(T^{-1}(\mathbf{u})) + \log |\det(\mathbf{J}_{T^{-1}}(\mathbf{u}))|] + \mathbb{H}(\psi) \quad (10.63)$$

This is a tractable objective, assuming the Jacobian is tractable, since the final entropy term is available in closed form, and we can use a Monte Carlo approximation of the expectation over \mathbf{u} .

Since the objective is stochastic, and reparameterizable, we can use SGD to optimize it. However, [Ing20] propose **deterministic ADVI**, in which the samples $\epsilon_s \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ are held fixed during the optimization process. This is called the common random numbers trick (Section 11.6.1), and makes the objective a deterministic function; this allows for the use of more powerful second-order optimization methods, such as BFGS. (Of course, if the dataset is large, we might need to use minibatch subsampling, which reintroduces stochasticity.)

10.2.2.2 Example: ADVI for beta-binomial model

To illustrate ADVI, we consider the 1d beta-binomial model from Section 7.4.4. We want to approximate $p(\theta|\mathcal{D})$ using the prior $p(\theta) = \text{Beta}(\theta|a, b)$ and likelihood $p(\mathcal{D}|\theta) = \prod_i \text{Ber}(y_i|\theta)$, where the sufficient statistics are $N_1 = 10$, $N_0 = 1$, and the prior is uninformative, $a = b = 1$. We use the transformation $\theta = T^{-1}(z) = \sigma(z)$, and optimize the ELBO with SGD. The results of this method are shown in Figure 7.4 and show that the Gaussian fit is a good approximation, despite the skewed nature of the posterior.

10.2.2.3 Example: ADVI for GMMs

In this section, we use ADVI to approximate the posterior of the parameters of a mixture of Gaussians. The difference from the VBEM algorithm of Section 10.3.6 is that we use ADVI combined with a Gaussian variational posterior, rather than using a mean field approximation defined by a product of conjugate distributions.

To apply ADVI, we marginalize out the discrete local discrete latents $m_n \in \{1, \dots, K\}$ analytically, so the likelihood has the form

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^N \left[\sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{y}_n | \boldsymbol{\mu}_k, \text{diag}(\boldsymbol{\Sigma}_k)) \right] \quad (10.64)$$

We use an uninformative Gaussian prior for the $\boldsymbol{\mu}_k$, a uniform LKJ prior for the \mathbf{L}_k , a log-normal prior for the $\boldsymbol{\sigma}_k$, and a uniform Dirichlet prior for the mixing weights $\boldsymbol{\pi}$. (See [Kuc+16, Fig 21] for a definition of the model in STAN syntax.) The posterior approximation for the unconstrained parameters is a block-diagonal gaussian. $q(\mathbf{u}) = \mathcal{N}(\mathbf{u} | \boldsymbol{\psi}_{\boldsymbol{\mu}}, \boldsymbol{\psi}_{\boldsymbol{\Sigma}})$, where the unconstrained parameters are computed using suitable bijections (see code for details).

We apply this method to the Old Faithful dataset from Figure 10.12, using $K = 10$ mixture components. The results are shown in Figure 10.7. In the top left, we show the special case where we constrain the posterior to be a MAP estimate, by setting $\boldsymbol{\psi}_{\boldsymbol{\Sigma}} = \mathbf{0}$. We see that there is no sparsity in the posterior, since there is no Bayesian “Occam factor” from marginalizing out the parameters. In panels c–d, we show 3 samples from the posterior. We see that the Bayesian method strongly prefers just 2 mixture components, although there is a small amount of support for some other Gaussian components (shown by the faint ellipses).

10.2.2.4 More complex posteriors

We can combine ADVI with any of the improved posterior approximations that we discuss in Section 10.4 — such as Gaussian mixtures [Mor+21b] or normalizing flows [ASD20] — to create a high-quality, automatic approximate inference scheme.

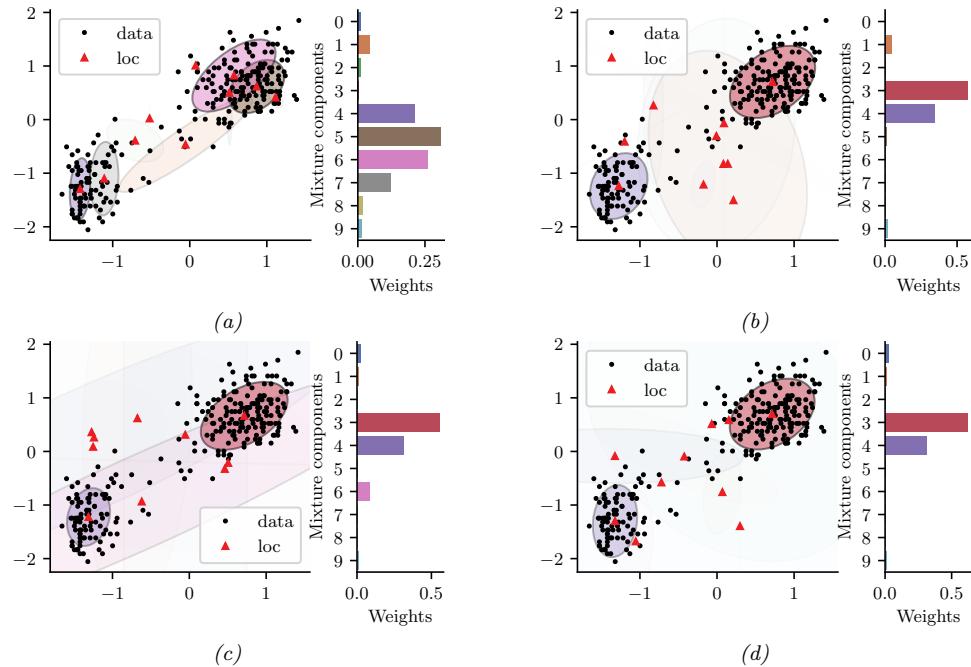


Figure 10.7: Posterior over the mixing weights (histogram) and the means and covariances of each Gaussian mixture component, using $K = 10$, when fitting the model to the Old Faithful dataset from Figure 10.12. (a) MAP approximation. (b-d) 3 samples from the Gaussian approximation. The intensity of the shading is proportional to the mixture weight. Generated by `gmm_advi_bijax.ipynb`.

10.2.3 Blackbox variational inference

In this section, we assume that we can evaluate $\tilde{\mathcal{L}}(\psi, z) = \log p(z, x) - \log q_\psi(z)$ pointwise, but we do not assume we can take gradients of this function. (For example, z may contain discrete variables.) We are thus treating the model as a “blackbox”. Hence this approach is called **blackbox variational inference** or **BBVI** [RGB14; ASD20].

10.2.3.1 Estimating the gradient using REINFORCE

To estimate the gradient of the ELBO, we will use the **score function estimator**, also called the **REINFORCE** estimator (Section 6.3.4). In particular, suppose we write the ELBO as

$$\hat{\mathcal{L}}(\psi) = \mathbb{E}_{q(z|\psi)} [\tilde{\mathcal{L}}(\psi, z)] = \mathbb{E}_{q(z|\psi)} [\log p(x, z) - \log q(z|\psi)] \quad (10.65)$$

Then from Equation (6.58) we have

$$\nabla_\psi \hat{\mathcal{L}}(\psi) = \mathbb{E}_{q(z|\psi)} [\tilde{\mathcal{L}}(\psi, z) \nabla_\psi \log q(z|\psi)] \quad (10.66)$$

We can then compute a Monte Carlo approximation to this:

$$\widehat{\nabla_{\psi} \mathcal{L}(\psi_t)} = \frac{1}{S} \sum_{s=1}^S \tilde{\mathcal{L}}(\psi, \mathbf{z}_s) \nabla_{\psi} \log q_{\psi}(\mathbf{z}_s) |_{\psi=\psi_t} \quad (10.67)$$

We can pass this to any kind of gradient optimizer, such as SGD or Adam.

10.2.3.2 Reducing the variance using control variates

In practice, the variance of this estimator is quite large, so it is important to use methods such as **control variates** or **CV** (Section 6.3.4.1). To see how this works, consider the naive gradient estimator in Equation (10.67), which for the i 'th component we can write as

$$\widehat{\nabla_{\psi_i} \mathcal{L}(\psi_t)}^{\text{naive}} = \frac{1}{S} \sum_{s=1}^S \tilde{g}_i(\mathbf{z}_s) \quad (10.68)$$

$$\tilde{g}_i(\mathbf{z}_s) = g_i(\mathbf{z}_s) \times \tilde{\mathcal{L}}(\psi, \mathbf{z}_s) \quad (10.69)$$

$$g_i(\mathbf{z}_s) = \nabla_{\psi_i} \log q_{\psi}(\mathbf{z}_s) \quad (10.70)$$

The control variate version of this can be obtained by replacing $\tilde{g}_i(\mathbf{z}_s)$ with

$$\tilde{g}_i^{cv}(\mathbf{z}) = \tilde{g}_i(\mathbf{z}) + c_i(\mathbb{E}[b_i(\mathbf{z})] - b_i(\mathbf{z})) \quad (10.71)$$

where $b_i(\mathbf{z})$ is a baseline function and c_i is some constant, to be specified below. A convenient baseline is the score function, $b_i(\mathbf{z}) = \nabla_{\psi_i} \log q_{\psi_i}(\mathbf{z}) = g_i(\mathbf{z})$, since this is correlated with $\tilde{g}_i(\mathbf{z})$, and has the property that $\mathbb{E}[b_i(\mathbf{z})] = \mathbf{0}$, since the expected value of the score function is zero, as we showed in Equation (3.44). Hence

$$\tilde{g}_i^{cv}(\mathbf{z}) = \tilde{g}_i(\mathbf{z}) - c_i g_i(\mathbf{z}) = g_i(\mathbf{z})(\tilde{\mathcal{L}}(\psi, \mathbf{z}) - c_i) \quad (10.72)$$

so the CV estimator is given by

$$\widehat{\nabla_{\psi_i} \mathcal{L}(\psi_t)}^{\text{cv}} = \frac{1}{S} \sum_{s=1}^S g_i(\mathbf{z}_s) \times (\tilde{\mathcal{L}}(\psi, \mathbf{z}_s) - c_i) \quad (10.73)$$

One can show that the optimal c_i that minimizes the variance of the CV estimator is

$$c_i = \frac{\text{Cov} [g_i(\mathbf{z}) \tilde{\mathcal{L}}(\psi, \mathbf{z}), g_i(\mathbf{z})]}{\mathbb{V}[g_i(\mathbf{z})]} \quad (10.74)$$

For more details, see e.g., [TND21].

10.3 Coordinate ascent VI

A common approximation in variational inference is to assume that all the latent variables are independent, i.e.,

$$q_{\psi}(\mathbf{z}) = \prod_{j=1}^J q_j(z_j) \quad (10.75)$$

where J is the number of hidden variables, and $q_j(z_j)$ is shorthand for $q_{\psi_j}(z_j)$, where ψ_j are the variational parameters for the j 'th distribution. This is called the **mean field** approximation.

From Equation (10.11), the ELBO becomes

$$\mathcal{L}(\psi) = \int q_{\psi}(\mathbf{z}) \log p_{\theta}(\mathbf{x}, \mathbf{z}) d\mathbf{z} + \sum_{j=1}^J \mathbb{H}(q_j) \quad (10.76)$$

since the entropy of a product distribution is the sum of entropies of each component in the product. The first term also often decomposes according to the Markov properties of the graphical model. This allows us to use a coordinate ascent optimization scheme to estimate each ψ_j , as we explain in Section 10.3.1. This is called **coordinate ascent variational inference** or **CAVI**, and is an alternative to gradient-based VI.

10.3.1 Derivation of CAVI algorithm

In this section, we derive the coordinate ascent variational inference (CAVI) procedure.

To derive the update equations, we initially assume there are just 3 discrete latent variables, to simplify notation. In this case the ELBO is given by

$$\mathcal{L}(q_1, q_2, q_3) = \sum_{z_1} \sum_{z_2} \sum_{z_3} q_1(z_1) q_2(z_2) q_3(z_3) \log \tilde{p}(z_1, z_2, z_3) + \sum_{j=1}^3 \mathbb{H}(q_j) \quad (10.77)$$

where we define $\tilde{p}(\mathbf{z}) = p_{\theta}(\mathbf{z}, \mathbf{x})$ for brevity. We will optimize this wrt each q_i , one at a time, keeping the others fixed.

Let us look at the objective for q_3 :

$$\mathcal{L}_3(q_3) = \sum_{z_3} q_3(z_3) \left[\sum_{z_1} \sum_{z_2} q_1(z_1) q_2(z_2) \log \tilde{p}(z_1, z_2, z_3) \right] + \mathbb{H}(q_3) + \text{const} \quad (10.78)$$

$$= \sum_{z_3} q_3(z_3) [g_3(z_3) - \log q_3(z_3)] + \text{const} \quad (10.79)$$

where

$$g_3(z_3) \triangleq \sum_{z_1} \sum_{z_2} q_1(z_1) q_2(z_2) \log \tilde{p}(z_1, z_2, z_3) = \mathbb{E}_{\mathbf{z}_{-3}} [\log \tilde{p}(z_1, z_2, z_3)] \quad (10.80)$$

where $\mathbf{z}_{-3} = (z_1, z_2)$ is all variables except z_3 . Here $g_3(z_3)$ can be interpreted as an expected negative energy (log probability). We can convert this into an unnormalized probability distribution by defining

$$\tilde{f}_3(z_3) = \exp(g_3(z_3)) \quad (10.81)$$

which we can normalize to get

$$f_3(z_3) = \frac{\tilde{f}_3(z_3)}{\sum_{z'_3} \tilde{f}_3(z'_3)} \propto \exp(g_3(z_3)) \quad (10.82)$$

Since $g_3(z_3) \propto \log f_3(z_3)$ we get

$$\mathbb{L}_3(q_3) = \sum_{z_3} q_3(z_3) [\log f_3(z_3) - \log g_3(z_3)] + \text{const} = -D_{\text{KL}}(q_3 \parallel f_3) + \text{const} \quad (10.83)$$

Since $D_{\text{KL}}(q_3 \parallel f_3)$ achieves its minimal value of 0 when $q_3(z_3) = f_3(z_3)$ for all z_3 , we see that $q_3^*(z_3) = f_3(z_3)$.

Now suppose that the joint distribution is defined by a Markov chain, where $z_1 \rightarrow z_2 \rightarrow z_3$, so $z_1 \perp z_3 | z_2$. Hence $\log \tilde{p}(z_1, z_2, z_3) = \log \tilde{p}(z_2, z_3 | z_1) + \log \tilde{p}(z_1)$, where the latter term is independent of $q_3(z_3)$. Thus the ELBO simplifies to

$$\mathbb{L}_3(q_3) = \sum_{z_3} q_3(z_3) \left[\sum_{z_2} q_2(z_2) \log \tilde{p}(z_2, z_3) \right] + \mathbb{H}(q_3) + \text{const} \quad (10.84)$$

$$= \sum_{z_3} q_3(z_3) [\log f_3(z_3) - \log g_3(z_3)] + \text{const} \quad (10.85)$$

where

$$f_3(z_3) \propto \exp \left[\sum_{z_2} q_2(z_2) \log \tilde{p}(z_2, z_3) \right] = \exp \left[\mathbb{E}_{\mathbf{z}_{\text{mb}_3}} [\log \tilde{p}(z_2, z_3)] \right] \quad (10.86)$$

where $\mathbf{z}_{\text{mb}_3} = z_2$ is the Markov blanket (Section 4.2.4.3) of z_3 . As before, the optimal variational distribution is given by $q_3(z_3) = f_3(z_3)$.

In general, when we have J groups of variables, the optimal variational distribution for the j 'th group is given by

$$q_j(\mathbf{z}_j) \propto \exp \left[\mathbb{E}_{\mathbf{z}_{\text{mb}_j}} [\log \tilde{p}(\mathbf{z}_j, \mathbf{z}_{\text{mb}_j})] \right] \quad (10.87)$$

(Compare to the equation for Gibbs sampling in Equation (12.19).) The CAVI method simply computes q_j for each dimension j in turn, in an iterative fashion (see Algorithm 10.4). Convergence is guaranteed since the bound is concave wrt each of the factors q_i [Bis06, p. 466].

Algorithm 10.4: Coordinate ascent variational inference (CAVI).

```

1 Initialize  $q_j(\mathbf{z}_j)$  for  $j = 1 : J$ 
2 foreach  $t = 1 : T$  do
3   foreach  $j = 1 : J$  do
4     Compute  $g_j(\mathbf{z}_j) = \mathbb{E}_{\mathbf{z}_{\text{mb}_j}} [\log \tilde{p}(\mathbf{z}_j, \mathbf{z}_{\text{mb}_j})]$ 
5     Compute  $q_j(\mathbf{z}_j) \propto \exp(g_j(\mathbf{z}_j))$ 

```

Note that the functional form of the q_i distributions does not need to be specified in advance, but will be determined by the form of the log joint. This is therefore called **free-form VI**, as opposed to fixed-form, where we explicitly choose a convenient distributional type for q (we discuss fixed-form VI in Section 10.2). We give some examples below that will make this clearer.

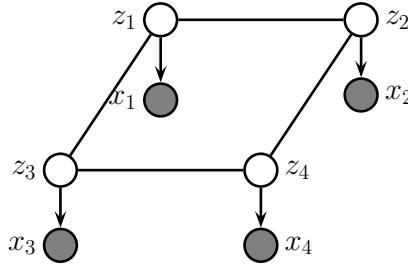


Figure 10.8: A grid-structured MRF with hidden nodes z_i and local evidence nodes x_i . The prior $p(\mathbf{z})$ is an undirected Ising model, and the likelihood $p(\mathbf{x}|\mathbf{z}) = \prod_i p(x_i|z_i)$ is a directed fully factored model.

10.3.2 Example: CAVI for the Ising model

In this section, we apply CAVI to perform mean field inference in an Ising model (Section 4.3.2.1), which is a kind of Markov random field defined on binary random variables, $z_i \in \{-1, +1\}$, arranged in a 2d grid.

Originally Ising models were developed as models of atomic spins for magnetic materials, although we will apply them to an image denoising problem. Specifically, let z_i be the hidden value of pixel i , and $x_i \in \mathbb{R}$ be the observed noisy value. See Figure 10.8 for the graphical model.

Let $L_i(z_i) \triangleq \log p(x_i|z_i)$ be the log likelihood for the i 'th pixel (aka the **local evidence** for node i in the graphical model). The overall likelihood has the form

$$p(\mathbf{x}|\mathbf{z}) = \prod_i p(x_i|z_i) = \exp\left(\sum_i L_i(z_i)\right) \quad (10.88)$$

Our goal is to approximate the posterior $p(\mathbf{z}|\mathbf{x})$. We will use an Ising model for the prior:

$$p(\mathbf{z}) = \frac{1}{Z_0} \exp(-\mathcal{E}_0(\mathbf{z})) \quad (10.89)$$

$$\mathcal{E}_0(\mathbf{z}) = - \sum_{i \sim j} W_{ij} z_i z_j \quad (10.90)$$

where we sum over each $i - j$ edge. Therefore the posterior has the form

$$p(\mathbf{z}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp(-\mathcal{E}(\mathbf{z})) \quad (10.91)$$

$$\mathcal{E}(\mathbf{z}) = \mathcal{E}_0(\mathbf{z}) - \sum_i L_i(z_i) \quad (10.92)$$

We will now make the following fully factored approximation:

$$q(\mathbf{z}) = \prod_i q_i(z_i) = \prod_i \text{Ber}(z_i|\mu_i) \quad (10.93)$$

where $\mu_i = \mathbb{E}_{q_i}[z_i]$ is the mean value of node i . To derive the update for the variational parameter μ_i , we first compute the unnormalized log joint, $\log \tilde{p}(\mathbf{z}) = -\mathcal{E}(\mathbf{z})$, dropping terms that do not involve

z_i :

$$\log \tilde{p}(\mathbf{z}) = z_i \sum_{j \in \text{nbr}_i} W_{ij} z_j + L_i(z_i) + \text{const} \quad (10.94)$$

This only depends on the states of the neighboring nodes. Hence

$$q_i(z_i) \propto \exp(\mathbb{E}_{q_{-i}(\mathbf{z})} [\log \tilde{p}(\mathbf{z})]) = \exp \left(z_i \sum_{j \in \text{nbr}_i} W_{ij} \mu_j + L_i(z_i) \right) \quad (10.95)$$

where $q_{-i}(\mathbf{z}) = \prod_{j \neq i} q(z_j)$. Thus we replace the states of the neighbors by their average values. (Note that this replaces binary variables with continuous ones.)

We now simplify this expression. Let $m_i = \sum_{j \in \text{nbr}_i} W_{ij} \mu_j$ be the mean field influence on node i . Also, let $L_i^+ \triangleq L_i(+1)$ and $L_i^- \triangleq L_i(-1)$. The approximate marginal posterior is given by

$$q_i(z_i = 1) = \frac{e^{m_i + L_i^+}}{e^{m_i + L_i^+} + e^{-m_i + L_i^-}} = \frac{1}{1 + e^{-2m_i + L_i^- - L_i^+}} = \sigma(2a_i) \quad (10.96)$$

$$a_i \triangleq m_i + 0.5(L_i^+ - L_i^-) \quad (10.97)$$

Similarly, we have $q_i(z_i = -1) = \sigma(-2a_i)$. From this we can compute the new mean for site i :

$$\mu_i = \mathbb{E}_{q_i} [z_i] = q_i(z_i = +1) \cdot (+1) + q_i(z_i = -1) \cdot (-1) \quad (10.98)$$

$$= \frac{1}{1 + e^{-2a_i}} - \frac{1}{1 + e^{2a_i}} = \frac{e^{a_i}}{e^{a_i} + e^{-a_i}} - \frac{e^{-a_i}}{e^{-a_i} + e^{a_i}} = \tanh(a_i) \quad (10.99)$$

We can turn the above equations into a fixed point algorithm by writing

$$\mu_i^t = \tanh \left(\sum_{j \in \text{nbr}_i} W_{ij} \mu_j^{t-1} + 0.5(L_i^+ - L_i^-) \right) \quad (10.100)$$

Following [MWJ99], we can use **damped updates** of the following form to improve convergence:

$$\mu_i^t = (1 - \lambda) \mu_i^{t-1} + \lambda \tanh \left(\sum_{j \in \text{nbr}_i} W_{ij} \mu_j^{t-1} + 0.5(L_i^+ - L_i^-) \right) \quad (10.101)$$

for $0 < \lambda < 1$. We can update all the nodes in parallel, or update them asynchronously.

Figure 10.9 shows the method in action, applied to a 2d Ising model with homogeneous attractive potentials, $W_{ij} = 1$. We use parallel updates with a damping factor of $\lambda = 0.5$. (If we don't use damping, we tend to get "checkerboard" artifacts.)

10.3.3 Variational Bayes

In Bayesian modeling, we treat the parameters $\boldsymbol{\theta}$ as latent variables. Thus our goal is to approximate the parameter posterior $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$. Applying VI to this problem is called **variational Bayes** [Att00].

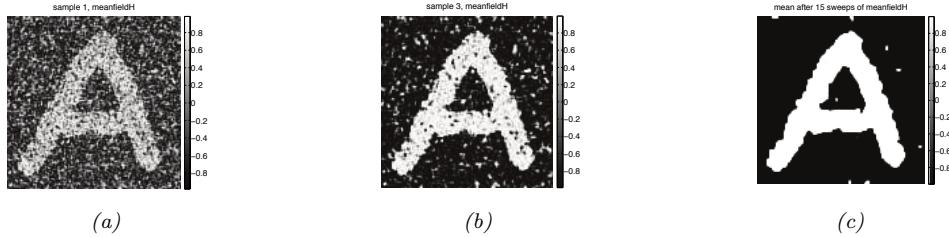


Figure 10.9: Example of image denoising using mean field (with parallel updates and a damping factor of 0.5). We use an Ising prior with $W_{ij} = 1$ and a Gaussian noise model with $\sigma = 2$. We show the results after 1, 3 and 15 iterations across the image. Compare to Figure 12.3, which shows the results of using Gibbs sampling. Generated by [ising_image_denoise_demo.ipynb](#).

In this section, we assume there are no latent variables except for the shared global parameters, so the model has the form

$$p(\boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathcal{D}_n | \boldsymbol{\theta}) \quad (10.102)$$

These conditional independencies are illustrated in Figure 10.5a.

We will fit the variational posterior by maximizing the ELBO

$$\mathcal{L}(\boldsymbol{\psi}_{\boldsymbol{\theta}} | \mathcal{D}) = \mathbb{E}_{q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}})} [\log p(\boldsymbol{\theta}, \mathcal{D})] + \mathbb{H}(q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}})) \quad (10.103)$$

We will assume the variational posterior factorizes over the parameters:

$$q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}}) = \prod_j q(\boldsymbol{\theta}_j | \boldsymbol{\psi}_{\theta_j}) \quad (10.104)$$

We can then update each $\boldsymbol{\psi}_{\theta_j}$ using CAVI (Section 10.3.1).

10.3.4 Example: VB for a univariate Gaussian

Consider inferring the parameters of a 1d Gaussian. The likelihood is given by $p(\mathcal{D} | \boldsymbol{\theta}) = \prod_{n=1}^N \mathcal{N}(x_n | \mu, \lambda^{-1})$, where μ is the mean and λ is the precision. Suppose we use a conjugate prior of the form

$$p(\mu, \lambda) = \mathcal{N}(\mu | \mu_0, (\kappa_0 \lambda)^{-1}) \text{Ga}(\lambda | a_0, b_0) \quad (10.105)$$

It is possible to derive the posterior $p(\mu, \lambda | \mathcal{D})$ for this model exactly, as shown in Section 3.4.3.3. However, here we use the VB method with the following factored approximate posterior:

$$q(\mu, \lambda) = q(\mu | \boldsymbol{\psi}_{\mu}) q(\lambda | \boldsymbol{\psi}_{\lambda}) \quad (10.106)$$

We do not need to specify the forms for the distributions $q(\mu | \boldsymbol{\psi}_{\mu})$ and $q(\lambda | \boldsymbol{\psi}_{\lambda})$; the optimal forms will “fall out” automatically during the derivation (and conveniently, they turn out to be Gaussian and gamma respectively). Our presentation follows [Mac03, p429].

10.3.4.1 Target distribution

The unnormalized log posterior has the form

$$\log \tilde{p}(\mu, \lambda) = \log p(\mu, \lambda, \mathcal{D}) = \log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda) + \log p(\lambda) \quad (10.107)$$

$$\begin{aligned} &= \frac{N}{2} \log \lambda - \frac{\lambda}{2} \sum_{n=1}^N (x_n - \mu)^2 - \frac{\kappa_0 \lambda}{2} (\mu - \mu_0)^2 \\ &\quad + \frac{1}{2} \log(\kappa_0 \lambda) + (a_0 - 1) \log \lambda - b_0 \lambda + \text{const} \end{aligned} \quad (10.108)$$

10.3.4.2 Updating $q(\mu|\psi_\mu)$

The optimal form for $q(\mu|\psi_\mu)$ is obtained by averaging over λ :

$$\log q(\mu|\psi_\mu) = \mathbb{E}_{q(\lambda|\psi_\lambda)} [\log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda)] + \text{const} \quad (10.109)$$

$$= -\frac{\mathbb{E}_{q(\lambda|\psi_\lambda)} [\lambda]}{2} \left\{ \kappa_0 (\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right\} + \text{const} \quad (10.110)$$

By completing the square one can show that $q(\mu|\psi_\mu) = \mathcal{N}(\mu|\mu_N, \kappa_N^{-1})$, where

$$\mu_N = \frac{\kappa_0 \mu_0 + N \bar{x}}{\kappa_0 + N}, \quad \kappa_N = (\kappa_0 + N) \mathbb{E}_{q(\lambda|\psi_\lambda)} [\lambda] \quad (10.111)$$

At this stage we don't know what $q(\lambda|\psi_\lambda)$ is, and hence we cannot compute $\mathbb{E}[\lambda]$, but we will derive this below.

10.3.4.3 Updating $q(\lambda|\psi_\lambda)$

The optimal form for $q(\lambda|\psi_\lambda)$ is given by

$$\log q(\lambda|\psi_\lambda) = \mathbb{E}_{q(\mu|\psi_\mu)} [\log p(\mathcal{D}|\mu, \lambda) + \log p(\mu|\lambda) + \log p(\lambda)] + \text{const} \quad (10.112)$$

$$\begin{aligned} &= (a_0 - 1) \log \lambda - b_0 \lambda + \frac{1}{2} \log \lambda + \frac{N}{2} \log \lambda \\ &\quad - \frac{\lambda}{2} \mathbb{E}_{q(\mu|\psi_\mu)} \left[\kappa_0 (\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right] + \text{const} \end{aligned} \quad (10.113)$$

We recognize this as the log of a gamma distribution, hence $q(\lambda|\psi_\lambda) = \text{Ga}(\lambda|a_N, b_N)$, where

$$a_N = a_0 + \frac{N+1}{2} \quad (10.114)$$

$$b_N = b_0 + \frac{1}{2} \mathbb{E}_{q(\mu|\psi_\mu)} \left[\kappa_0 (\mu - \mu_0)^2 + \sum_{n=1}^N (x_n - \mu)^2 \right] \quad (10.115)$$

10.3.4.4 Computing the expectations

To implement the updates, we have to specify how to compute the various expectations. Since $q(\mu) = \mathcal{N}(\mu | \mu_N, \kappa_N^{-1})$, we have

$$\mathbb{E}_{q(\mu)} [\mu] = \mu_N \tag{10.116}$$

$$\mathbb{E}_{q(\mu)} [\mu^2] = \frac{1}{\kappa_N} + \mu_N^2 \tag{10.117}$$

Since $q(\lambda) = \text{Ga}(\lambda | a_N, b_N)$, we have

$$\mathbb{E}_{q(\lambda)} [\lambda] = \frac{a_N}{b_N} \tag{10.118}$$

We can now give explicit forms for the update equations. For $q(\mu)$ we have

$$\mu_N = \frac{\kappa_0 \mu_0 + N \bar{x}}{\kappa_0 + N} \tag{10.119}$$

$$\kappa_N = (\kappa_0 + N) \frac{a_N}{b_N} \tag{10.120}$$

and for $q(\lambda)$ we have

$$a_N = a_0 + \frac{N + 1}{2} \tag{10.121}$$

$$b_N = b_0 + \frac{1}{2} \kappa_0 (\mathbb{E} [\mu^2] + \mu_0^2 - 2\mathbb{E} [\mu] \mu_0) + \frac{1}{2} \sum_{n=1}^N (x_n^2 + \mathbb{E} [\mu^2] - 2\mathbb{E} [\mu] x_n) \tag{10.122}$$

We see that μ_N and a_N are in fact fixed constants, and only κ_N and b_N need to be updated iteratively. (In fact, one can solve for the fixed points of κ_N and b_N analytically, but we don't do this here in order to illustrate the iterative updating scheme.)

10.3.4.5 Illustration

Figure 10.10 gives an example of this method in action. The green contours represent the exact posterior, which is Gaussian-gamma. The dotted red contours represent the variational approximation over several iterations. We see that the final approximation is reasonably close to the exact solution. However, it is more "compact" than the true distribution. It is often the case that mean field inference underestimates the posterior uncertainty, for reasons explained in Section 5.1.4.1.

10.3.4.6 Lower bound

In VB, we maximize a lower bound on the log marginal likelihood:

$$\mathbb{L}(\psi_\theta | \mathcal{D}) \leq \log p(\mathcal{D}) = \log \iint p(\mathcal{D} | \mu, \lambda) p(\mu, \lambda) d\mu d\lambda \tag{10.123}$$

It is very useful to compute the lower bound itself, for three reasons. First, it can be used to assess convergence of the algorithm. Second, it can be used to assess the correctness of one's code: as with

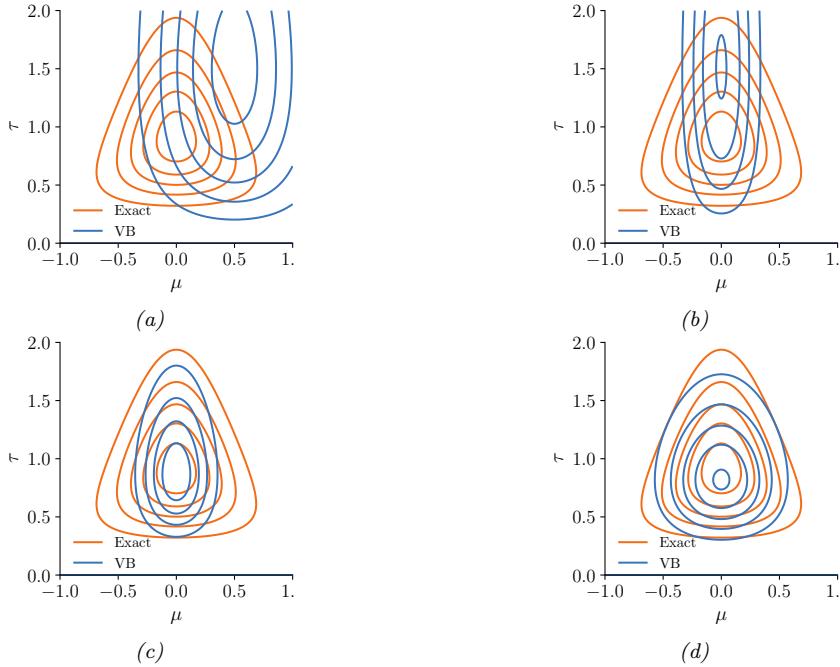


Figure 10.10: Factored variational approximation (orange) to the Gaussian-gamma distribution (blue). (a) Initial guess. (b) After updating $q(\mu|\psi_\mu)$. (c) After updating $q(\lambda|\psi_\lambda)$. (d) At convergence (after 5 iterations). Adapted from Fig. 10.4 of [Bis06]. Generated by [unigauss_vb_demo.ipynb](#).

EM, if we use CAVI to optimize the objective, the bound should increase monotonically at each iteration, otherwise there must be a bug. Third, the bound can be used as an approximation to the marginal likelihood, which can be used for Bayesian model selection or empirical Bayes (see Section 10.1.3). In the case of the current model, one can show that the lower bound has the following form:

$$\bar{L} = \text{const} + \frac{1}{2} \ln \frac{1}{\kappa_N} + \ln \Gamma(a_N) - a_N \ln b_N \quad (10.124)$$

10.3.5 Variational Bayes EM

In Bayesian latent variable models, we have two forms of hidden variables: local (or per example) hidden variables \mathbf{z}_n , and global (shared) hidden variables $\boldsymbol{\theta}$, which represent the parameters of the model. See Figure 10.5b for an illustration. (Note that the parameters, which are fixed in number, are sometimes called **intrinsic variables**, whereas the local hidden variables are called **extrinsic variables**.) If $\mathbf{h} = (\boldsymbol{\theta}, \mathbf{z}_{1:N})$ represents all the hidden variables, then the joint distribution is given by

$$p(\mathbf{h}, \mathcal{D}) = p(\boldsymbol{\theta}, \mathbf{z}_{1:N}, \mathcal{D}) = p(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathbf{z}_n | \boldsymbol{\theta}) p(\mathbf{x}_n | \mathbf{z}_n, \boldsymbol{\theta}) \quad (10.125)$$

We will make the following mean field assumption:

$$q(\boldsymbol{\theta}, \mathbf{z}_{1:N} | \boldsymbol{\psi}_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}}) = q(\boldsymbol{\theta} | \boldsymbol{\psi}_{\boldsymbol{\theta}}) \prod_{n=1}^N q(\mathbf{z}_n | \boldsymbol{\psi}_n) \quad (10.126)$$

where $\boldsymbol{\psi} = (\boldsymbol{\psi}_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}})$.

We will use VI to maximize the ELBO:

$$\mathcal{L}(\boldsymbol{\psi} | \mathcal{D}) = \mathbb{E}_{q(\boldsymbol{\theta}, \mathbf{z}_{1:N} | \boldsymbol{\psi}_{1:N}, \boldsymbol{\psi}_{\boldsymbol{\theta}})} [\log p(\mathbf{z}_{1:N}, \boldsymbol{\theta}, \mathcal{D}) - \log q(\boldsymbol{\theta}, \mathbf{z}_{1:N})] \quad (10.127)$$

If we use the mean field assumption, then we can apply the CAVI approach to optimize each set of variational parameters. In particular, we can alternate between optimizing the $q_n(\mathbf{z}_n)$ in parallel, independently of each other, with $q(\boldsymbol{\theta})$ held fixed, and then optimizing $q(\boldsymbol{\theta})$ with the q_n held fixed. This is known as **variational Bayes EM** [BG06]. It is similar to regular EM, except in the E step, we infer an approximate posterior for \mathbf{z}_n averaging out the parameters (instead of plugging in a point estimate), and in the M step, we update the parameter posterior parameters using the expected sufficient statistics.

Now suppose we approximate $q(\boldsymbol{\theta})$ by a delta function, $q(\boldsymbol{\theta}) = \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})$. The Bayesian LVM ELBO objective from Equation (10.127) simplifies to the “LVM ELBO”:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\psi}_{1:N} | \mathcal{D}) = \mathbb{E}_{q(\mathbf{z}_{1:N} | \boldsymbol{\psi}_{1:N})} [\log p(\boldsymbol{\theta}, \mathcal{D}, \mathbf{z}_{1:N}) - \log q(\mathbf{z}_{1:N} | \boldsymbol{\psi}_{1:N})] \quad (10.128)$$

We can optimize this using the **variational EM** algorithm, which is a CAVI algorithm which updates the $\boldsymbol{\psi}_n$ in parallel in the variational E step, and then updates $\boldsymbol{\theta}$ in the M step.

VEM is simpler than VBEM since in the variational E step, we compute $q(\mathbf{z}_n | \mathbf{x}_n, \hat{\boldsymbol{\theta}})$, instead of $\mathbb{E}_{\boldsymbol{\theta}}[q(\mathbf{z}_n | \mathbf{x}_n, \boldsymbol{\theta})]$; that is, we plug in a point estimate of the model parameters, rather than averaging over the parameters. For more details on VEM, see Section 10.1.3.

10.3.6 Example: VBEM for a GMM

Consider a standard Gaussian mixture model (GMM):

$$p(\mathbf{z}, \mathbf{x} | \boldsymbol{\theta}) = \prod_n \prod_k \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1})^{z_{nk}} \quad (10.129)$$

where $z_{nk} = 1$ if datapoint n belongs to cluster k , and $z_{nk} = 0$ otherwise. Our goal is to approximate the posterior $p(\mathbf{z}, \boldsymbol{\theta} | \mathbf{x})$ under the following conjugate prior

$$p(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \check{\boldsymbol{\alpha}}) \prod_k \mathcal{N}(\boldsymbol{\mu}_k | \check{\mathbf{m}}, (\check{\kappa} \boldsymbol{\Lambda}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \check{\mathbf{L}}, \check{\nu}) \quad (10.130)$$

where $\boldsymbol{\Lambda}_k$ is the precision matrix for cluster k . For the mixing weights, we usually use a symmetric prior, $\check{\boldsymbol{\alpha}} = \alpha_0 \mathbf{1}$.

The exact posterior $p(\mathbf{z}, \boldsymbol{\theta} | \mathcal{D})$ is a mixture of K^N distributions, corresponding to all possible labelings \mathbf{z} , which is intractable to compute. In this section, we derive a VBEM algorithm, which will approximate the posterior around a local mode. We follow the presentation of [Bis06, Sec 10.2]. (See also Section 10.2.1.5 and Section 10.2.2.3, where we discuss variational approximations based on stochastic gradient descent, which can scale better to large datasets compared to VBEM.)

10.3.6.1 The variational posterior

We will use the standard mean field approximation to the posterior: $q(\boldsymbol{\theta}, \mathbf{z}_{1:N}) = q(\boldsymbol{\theta}) \prod_n q_n(\mathbf{z}_n)$. At this stage we have not specified the forms of the q functions; these will be determined by the form of the likelihood and prior. Below we will show that the optimal forms are as follows:

$$q_n(z_n) = \text{Cat}(\mathbf{z}_n | \mathbf{r}_n) \quad (10.131)$$

$$q(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \hat{\boldsymbol{\alpha}}) \prod_k \mathcal{N}(\boldsymbol{\mu}_k | \hat{\mathbf{m}}_k, (\hat{\kappa}_k \boldsymbol{\Lambda}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \hat{\mathbf{L}}_k, \hat{\nu}_k) \quad (10.132)$$

where \mathbf{r}_n are the posterior responsibilities, and the parameters with hats on them are the hyperparameters from the prior updated with data.

10.3.6.2 Derivation of $q(\boldsymbol{\theta})$ (variational M step)

Using the mean field recipe in Algorithm 10.4, we write down the log joint, and take expectations over all variables except $\boldsymbol{\theta}$, so we average out the \mathbf{z}_n wrt $q(\mathbf{z}_n) = \text{Cat}(z_n | \mathbf{r}_n)$:

$$\begin{aligned} \log q(\boldsymbol{\theta}) &= \log p(\boldsymbol{\pi}) + \underbrace{\sum_n \mathbb{E}_{q(z_n)} [\log p(\mathbf{z}_n | \boldsymbol{\pi})]}_{L_{\boldsymbol{\pi}}} \\ &\quad + \sum_k \left[\underbrace{\log p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) \sum_n \mathbb{E}_{q(z_n)} [z_{nk}] \log \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1})}_{L_{\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k}} \right] + \text{const} \end{aligned} \quad (10.133)$$

Since the expected log joint factorizes into a term involving $\boldsymbol{\pi}$ and terms involving $(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)$, we see that the variational posterior also factorizes into the form

$$q(\boldsymbol{\theta}) = q(\boldsymbol{\pi}) \prod_k q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) \quad (10.134)$$

For the $\boldsymbol{\pi}$ term, we have

$$\log q(\boldsymbol{\pi}) = (\alpha_0 - 1) \sum_k \log \pi_k + \sum_k \sum_n r_{nk} \log \pi_k + \text{const} \quad (10.135)$$

Exponentiating, we recognize this as a Dirichlet distribution:

$$q(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi} | \hat{\boldsymbol{\alpha}}) \quad (10.136)$$

$$\hat{\alpha}_k = \alpha_0 + N_k \quad (10.137)$$

$$N_k = \sum_n r_{nk} \quad (10.138)$$

For the $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ terms, we have

$$q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = \mathcal{N}(\boldsymbol{\mu}_k | \widehat{\boldsymbol{m}}_k, (\widehat{\kappa}_k \boldsymbol{\Lambda}_k)^{-1}) \text{Wi}(\boldsymbol{\Lambda}_k | \widehat{\mathbf{L}}_k, \widehat{\nu}_k) \quad (10.139)$$

$$\widehat{\kappa}_k = \check{\kappa} + N_k \quad (10.140)$$

$$\widehat{\boldsymbol{m}}_k = (\check{\kappa} \check{\boldsymbol{m}} + N_k \bar{\boldsymbol{x}}_k) / \widehat{\kappa}_k \quad (10.141)$$

$$\widehat{\mathbf{L}}_k^{-1} = \check{\mathbf{L}}^{-1} + N_k \mathbf{S}_k + \frac{\check{\kappa} N_k}{\check{\kappa} + N_k} (\bar{\boldsymbol{x}}_k - \check{\boldsymbol{m}})(\bar{\boldsymbol{x}}_k - \check{\boldsymbol{m}})^\top \quad (10.142)$$

$$\widehat{\nu}_k = \check{\nu} + N_k \quad (10.143)$$

$$\bar{\boldsymbol{x}}_k = \frac{1}{N_k} \sum_n r_{nk} \boldsymbol{x}_n \quad (10.144)$$

$$\mathbf{S}_k = \frac{1}{N_k} \sum_n r_{nk} (\boldsymbol{x}_n - \bar{\boldsymbol{x}}_k)(\boldsymbol{x}_n - \bar{\boldsymbol{x}}_k)^\top \quad (10.145)$$

This is very similar to the M step for MAP estimation for GMMs, except here we are computing the parameters of the posterior for $\boldsymbol{\theta}$ rather than a point estimate $\hat{\boldsymbol{\theta}}$.

10.3.6.3 Derivation of $q(z)$ (variational E step)

The variational E step is more interesting, since it is quite different from the E step in regular EM, because we need to average over the parameters, rather than condition on them. In particular, we have

$$\begin{aligned} \log q(\boldsymbol{z}) &= \sum_n \sum_k z_{nk} \left(\mathbb{E}_{q(\boldsymbol{\pi})} [\log \pi_k] + \frac{1}{2} \mathbb{E}_{q(\boldsymbol{\Lambda}_k)} [\log |\boldsymbol{\Lambda}_k|] - \frac{D}{2} \log(2\pi) \right. \\ &\quad \left. - \frac{1}{2} \mathbb{E}_{q(\boldsymbol{\theta})} [(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Lambda}_k (\boldsymbol{x}_n - \boldsymbol{\mu}_k)] \right) + \text{const} \end{aligned} \quad (10.146)$$

Using the fact that $q(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi} | \widehat{\boldsymbol{\alpha}})$, one can show that

$$\exp(\mathbb{E}_{q(\boldsymbol{\pi})} [\log \pi_k]) = \frac{\exp(\psi(\widehat{\alpha}_k))}{\exp(\psi(\sum_{k'} \widehat{\alpha}_{k'}))} \triangleq \tilde{\pi}_k \quad (10.147)$$

where ψ is the **digamma function**:

$$\psi(x) = \frac{d}{dx} \log \Gamma(x) \quad (10.148)$$

This takes care of the first term.

For the second term, one can show

$$\mathbb{E}_{q(\boldsymbol{\Lambda}_k)} [\log |\boldsymbol{\Lambda}_k|] = \sum_{j=1}^D \psi\left(\frac{\widehat{\nu}_k + 1 - j}{2}\right) + D \log 2 + \log |\widehat{\mathbf{L}}_k| \quad (10.149)$$

Finally, for the expected value of the quadratic form, one can show

$$\mathbb{E}_{q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)} [(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top \boldsymbol{\Lambda}_k (\boldsymbol{x}_n - \boldsymbol{\mu}_k)] = D \widehat{\kappa}_k^{-1} + \widehat{\nu}_k (\boldsymbol{x}_n - \widehat{\boldsymbol{m}}_k)^\top \widehat{\mathbf{L}}_k (\boldsymbol{x}_n - \widehat{\boldsymbol{m}}_k) \triangleq \tilde{\Lambda}_k \quad (10.150)$$

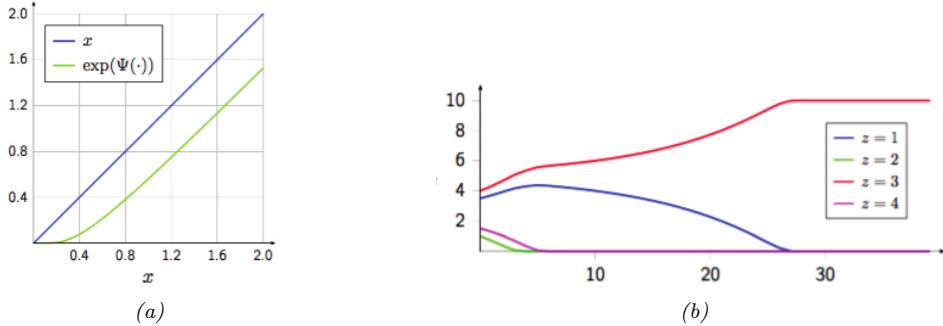


Figure 10.11: (a) We plot $\exp(\psi(x))$ vs x . We see that this function performs a form of shrinkage, so that small values get set to zero. (b) We plot N_k vs time for 4 different states (z values), starting from random initial values. We perform a series of VBEM updates, ignoring the likelihood term. We see that states that initially had higher counts get reinforced, and sparsely populated states get killed off. From [LK07]. Used with kind permission of Percy Liang.

Thus we get that the posterior responsibility of cluster k for datapoint n is

$$r_{nk} \propto \tilde{\pi}_k \tilde{\Lambda}_k^{\frac{1}{2}} \exp\left(-\frac{D}{2 \tilde{\kappa}_k} - \frac{\hat{\nu}_k}{2} (\mathbf{x}_n - \hat{\mathbf{m}}_k)^T \tilde{\Lambda}_k (\mathbf{x}_n - \hat{\mathbf{m}}_k)\right) \quad (10.151)$$

Compare this to the expression used in regular EM:

$$r_{nk}^{EM} \propto \hat{\pi}_k |\hat{\Lambda}_k|^{\frac{1}{2}} \exp\left(-\frac{1}{2} (\mathbf{x}_n - \hat{\mu}_k)^T \hat{\Lambda}_k (\mathbf{x}_n - \hat{\mu}_k)\right) \quad (10.152)$$

where $\hat{\pi}_k$ is the MAP estimate for π_k . The significance of this difference is discussed in Section 10.3.6.4.

10.3.6.4 Automatic sparsity inducing effects of VBEM

In regular EM, the E step has the form given in Equation (10.152), whereas in VBEM, the E step has the form given in Equation (10.151). Although they look similar, they differ in an important way. To understand this, let us ignore the likelihood term, and just focus on the prior. From Equation (10.147) we have

$$r_{nk}^{VB} = \tilde{\pi}_k = \frac{\exp(\psi(\hat{\alpha}_k))}{\exp(\psi(\sum_{k'} \hat{\alpha}_{k'}))} \quad (10.153)$$

And from the usual EM MAP estimation equations for GMM mixing weights (see e.g., [Mur22, Sec 8.7.3.4]) we have

$$r_{nk}^{EM} = \hat{\pi}_k = \frac{\hat{\alpha}_k - 1}{\sum_{k'} (\hat{\alpha}_{k'} - 1)} \quad (10.154)$$

where $\hat{\alpha}_k = \alpha_0 + N_k$, and $N_k = \sum_n r_{nk}$ is the expected number of assignments to cluster k .

We know from Figure 2.6 that using $\alpha_0 \ll 1$ causes $\boldsymbol{\pi}$ to be sparse, which will encourage \mathbf{r}_n to be sparse, which will “kill off” unnecessary mixture components (i.e., ones for which $N_k \ll N$, meaning very few datapoints are assigned to cluster k). To encourage this sparsity promoting effect, let us set $\alpha_0 = 0$. In this case, the updated parameters for the mixture weights are given by the following:

$$\tilde{\pi}_k = \frac{\exp(\psi(N_k))}{\exp(\psi(\sum_{k'} N_{k'}))} \quad (10.155)$$

$$\hat{\pi}_k = \frac{N_k - 1}{\sum_{k'} (N_{k'} - 1)} \quad (10.156)$$

Now consider a cluster which has no assigned data, so $N_k = 0$. In regular EM, $\hat{\pi}_k$ might end up negative, as pointed out in [FJ02]. (This will not occur if we use maximum likelihood training, which corresponds to $\alpha_0 = 1$, but this will not induce any sparsity, either.) This problem does not arise in VBEM, since we use the digamma function, which is always positive, as shown in Figure 10.11(a).

More interestingly, let us consider the effect of these updates on clusters that have unequal, but non-zero, number of assignments. Suppose we start with a random assignment of counts to 4 clusters, and iterate the VBEM algorithm, ignoring the contribution from the likelihood for simplicity. Figure 10.11(b) shows how the counts N_k evolve over time. We notice that clusters that started out with small counts end up with zero counts, and clusters that started out with large counts end up with even larger counts. In other words, the initially popular clusters get more and more members. This is called the **rich get richer** phenomenon; we will encounter it again in Supplementary Section 31.2, when we discuss Dirichlet process mixture models.

The reason for this effect is shown in Figure 10.11(a): we see that $\exp(\psi(N_k)) < N_k$, and is zero if N_k is sufficiently small, similar to the soft-thresholding behavior induced by ℓ_1 -regularization (see Section 15.2.6). Importantly, this effect of reducing N_k is greater on clusters with small counts.

We now demonstrate this automatic pruning method on a real example. We fit a mixture of 6 Gaussians to the Old Faithful dataset, using $\alpha_0 = 0.001$. Since the data only really “needs” 2 clusters, the remaining 4 get “killed off”, as shown in Figure 10.12. In Figure 10.13, we plot the initial and final values of α_k ; we see that $\hat{\alpha}_k = 0$ for all but two of the components k .

Thus we see that VBEM for GMMs with a sparse Dirichlet prior provides an efficient way to choose the number of clusters. Similar techniques can be used to choose the number of states in an HMM and other latent variable models. However, this **variational pruning effect** (also called **posterior collapse**), is not always desirable, since it can cause the model to “ignore” the latent variables \mathbf{z} if the likelihood function $p(\mathbf{x}|\mathbf{z})$ is sufficiently powerful. We discuss this more in Section 21.4.

10.3.6.5 Lower bound on the marginal likelihood

The VBEM algorithm is maximizing the following lower bound

$$\mathcal{L} = \sum_{\mathbf{z}} \int d\boldsymbol{\theta} q(\mathbf{z}, \boldsymbol{\theta}) \log \frac{p(\mathbf{x}, \mathbf{z}, \boldsymbol{\theta})}{q(\mathbf{z}, \boldsymbol{\theta})} \leq \log p(\mathbf{x}) \quad (10.157)$$

This quantity increases monotonically with each iteration, as shown in Figure 10.14.

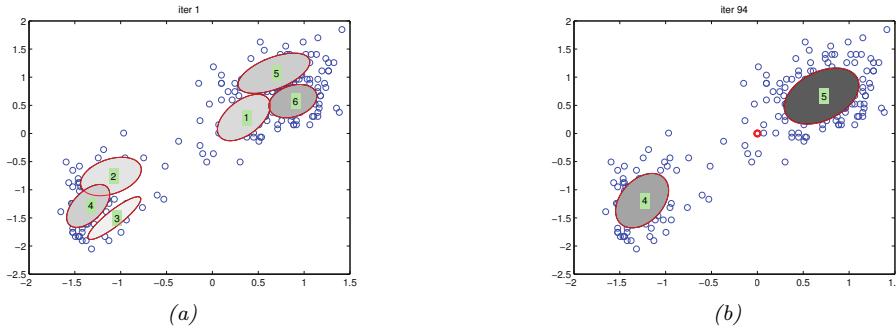


Figure 10.12: We visualize the posterior mean parameters at various stages of the VBEM algorithm applied to a mixture of Gaussians model on the Old Faithful data. Shading intensity is proportional to the mixing weight. We initialize with K-means and use $\alpha_0 = 0.001$ as the Dirichlet hyper-parameter. (The red dot on the right panel represents all the unused mixture components, which collapse to the prior at 0.) Adapted from Figure 10.6 of [Bis06]. Generated by `gmm_vb_em.ipynb`.

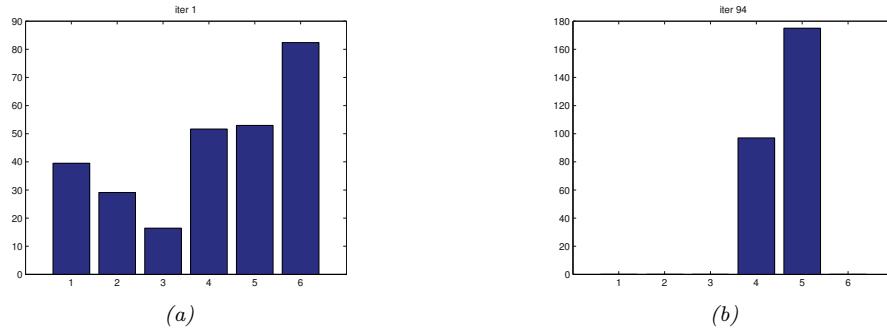


Figure 10.13: We visualize the posterior values of α_k for the model in Figure 10.12 after the first and last iteration of the algorithm. We see that unnecessary components get “killed off”. (Interestingly, the initially large cluster 6 gets “replaced” by cluster 5.) Generated by [gmm_vb_em.ipynb](#).

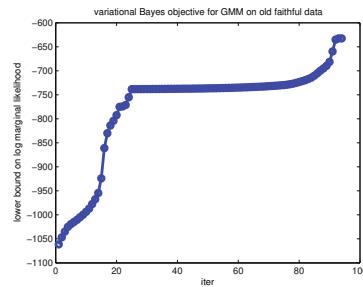


Figure 10.14: Lower bound vs iterations for the VB algorithm in Figure 10.12. The steep parts of the curve correspond to places where the algorithm figures out that it can increase the bound by “killing off” unnecessary mixture components, as described in Section 10.3.6.6. The plateaus correspond to slowly moving the clusters around. Generated by `gmm vb em.ipynb`.

10.3.6.6 Model selection using VBEM

Section 10.3.6.4 discusses a way to choose K automatically, during model fitting, by “killing off” unneeded clusters. An alternative approach is to fit several models, and then to use the variational lower bound to the log marginal likelihood, $\mathcal{L}(K) \leq \log p(\mathcal{D}|K)$, to approximate $p(K|\mathcal{D})$. In particular, if we have a uniform prior, we get the posterior

$$p(K|\mathcal{D}) = \frac{p(\mathcal{D}|K)}{\sum_{K'} p(\mathcal{D}|K')} \approx \frac{e^{\mathcal{L}(K)}}{\sum_{K'} e^{\mathcal{L}(K')}} \quad (10.158)$$

It is shown in [BG06] that the VB approximation to the marginal likelihood is more accurate than BIC [BG06]. However, the lower bound needs to be modified somewhat to take into account the lack of identifiability of the parameters. In particular, although VB will approximate the volume occupied by the parameter posterior, it will only do so around one of the local modes. With K components, there are $K!$ equivalent modes, which differ merely by permuting the labels. Therefore a more accurate approximation to the log marginal likelihood is to use $\log p(\mathcal{D}|K) \approx \mathcal{L}(K) + \log(K!)$.

10.3.7 Variational message passing (VMP)

In this section, we describe the CAVI algorithm for a generic model in which each complete conditional, $p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x})$, is in the exponential family, i.e.,

$$p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x}) = h(\mathbf{z}_j) \exp[\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})^\top \mathcal{T}(\mathbf{z}_j) - A_j(\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x}))] \quad (10.159)$$

where $\mathcal{T}(\mathbf{z}_j)$ is the vector of sufficient statistics, $\boldsymbol{\eta}_j$ are the natural parameters, A_j is the log partition function, and $h(\mathbf{z}_j)$ is the base distribution. This assumption holds if the prior $p(\mathbf{z}_j)$ is conjugate to the likelihood, $p(\mathbf{z}_{-j}, \mathbf{x}|\mathbf{z}_j)$.

If Equation (10.159) holds, the mean field update node j becomes

$$q_j(\mathbf{z}_j) \propto \exp [\mathbb{E} [\log p(\mathbf{z}_j|\mathbf{z}_{-j}, \mathbf{x})]] \quad (10.160)$$

$$= \exp \left[\log h(\mathbf{z}_j) + \mathbb{E} [\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})]^\top \mathcal{T}(\mathbf{z}_j) - \mathbb{E} [A_j(\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x}))] \right] \quad (10.161)$$

$$\propto h(\mathbf{z}_j) \exp \left[\mathbb{E} [\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})]^\top \mathcal{T}(\mathbf{z}_j) \right] \quad (10.162)$$

Thus we update the local natural parameters using the expected values of the other nodes. These become the new variational parameters:

$$\boldsymbol{\psi}_j = \mathbb{E} [\boldsymbol{\eta}_j(\mathbf{z}_{-j}, \mathbf{x})] \quad (10.163)$$

We can generalize the above approach to work with any model where each full conditional is conjugate. The resulting algorithm is known as **variational message passing** or **VMP** [WB05] that works for any directed graphical model. VMP is similar to belief propagation (Section 9.3): at each iteration, each node collects all the messages from its parents, and all the messages from its children (which might require the children to get messages from their co-parents), and combines them to compute the expected value of the node’s sufficient statistics. The messages that are sent are the expected sufficient statistics of a node, rather than just a discrete or Gaussian distribution (as in BP). Several software libraries have implemented this framework (see e.g., [Win; Min+18; Lut16; Wan17]).

VMP can be extended to the case where each full conditional is conditionally conjugate using the CVI framework in [Supplementary](#) Section 10.3.1. See also [ABV21], where they use local Laplace approximations to intractable factors inside of a message passing framework.

10.3.8 Autoconj

The VMP method requires the user to manually specify a graphical model; the corresponding node update equations are then computed for each node using a lookup table, for each possible combination of node types. It is possible to automatically derive these update equations for any conditionally conjugate directed graphical model using a technique called **autoconj** [HJT18]. This is analogous to the use of automatic differentiation (autodiff) to derive the gradient for any differentiable function. (Note that autoconj uses autodiff internally.) The resulting full conditionals can be used for CAVI, and also for Gibbs sampling (Section 12.3).

10.4 More accurate variational posteriors

In general, we can improve the tightness of the ELBO lower bound, and hence reduce the KL divergence of our posterior approximation, if we use more flexible posterior families (although optimizing within more flexible families may be slower, and can incur statistical error if the sample size is low [Bha+21]). In this section, we give several examples of more accurate variational posteriors, going beyond fully factored mean field approximations, or simple unimodal Gaussian approximations.

10.4.1 Structured mean field

The mean field assumption is quite strong, and can sometimes give poor results. Fortunately, sometimes we can exploit **tractable substructure** in our problem, so that we can efficiently handle some kinds of dependencies between the variables in the posterior in an analytic way, rather than assuming they are all independent. This is called the **structured mean field** approach [SJ95].

A common example arises when applying VI to time series models, such as HMMs, where the latent variables within each sequence are usually highly correlated across time. Rather than assuming a fully factorized posterior, we can treat each sequence $\mathbf{z}_{n,1:T}$ as a block, and just assume independence between blocks and the parameters: $q(\mathbf{z}_{1:N,1:T}, \boldsymbol{\theta}) = q(\boldsymbol{\theta}) \prod_{n=1}^N q(\mathbf{z}_{n,1:T})$, where $q(\mathbf{z}_{n,1:T}) = \prod_t q(\mathbf{z}_{n,t} | \mathbf{z}_{n,t-1})$. We can compute the joint distribution $q(\mathbf{z}_{n,1:T})$, taking into account the dependence between time steps, using the forwards-backwards algorithm. For details, see [JW14; Fot+14]. A similar approach was applied to the factorial HMM model, as we discuss in [Supplementary](#) Section 10.3.2.

An automatic way to derive a structured variational approximation to a probabilistic model, specified by a probabilistic programming language, is discussed in [AHG20].

10.4.2 Hierarchical (auxiliary variable) posteriors

Suppose $q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x}) = \prod_k q_{\boldsymbol{\phi}}(z_k | \mathbf{x})$ is a factorized distribution, such as a diagonal Gaussian. This does not capture dependencies between the latent variables (components of \mathbf{z}). We could of course use a full covariance matrix, but this might be too expensive.

An alternative approach is to use a hierarchical model, in which we add **auxiliary latent variables** \mathbf{a} , which are used to increase the flexibility of the variational posterior. In particular, we can still assume $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a})$ is conditionally factorized, but when we marginalize out \mathbf{a} , we induce dependencies between the elements of \mathbf{z} , i.e.,

$$q_\phi(\mathbf{z}|\mathbf{x}) = \int q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a})q_\phi(\mathbf{a}|\mathbf{x})d\mathbf{a} \neq \prod_k q_\phi(z_k|\mathbf{x}) \quad (10.164)$$

This is called a **hierarchical variational model** [Ran16], or an **auxiliary variable deep generative model** [Maa+16].

In [TRB16], they model $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{a})$ as a Gaussian process, which is a flexible nonparametric distribution (see Chapter 18), where \mathbf{a} are the inducing points. This combination is called a **variational GP**.

10.4.3 Normalizing flow posteriors

Normalizing flows are a class of probability models which work by passing a simple source distribution, such as a diagonal Gaussian, through a series of nonlinear, but invertible, mappings f to create a more complex distribution. This can be used to get more accurate posterior approximations than standard Gaussian VI, as we discuss in Section 23.1.2.2.

10.4.4 Implicit posteriors

In Chapter 26, we discuss implicit probability distributions, which are models which we can sample from, but which we cannot evaluate pointwise. For example, consider passing a Gaussian noise term, $\mathbf{z}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, through a nonlinear, *non-invertible* mapping f to create $\mathbf{z} = f(\mathbf{z}_0)$; it is easy to sample from $q(\mathbf{z})$, but it is intractable to evaluate the density $q(\mathbf{z})$ (unlike with flows). This makes it hard to evaluate the log density ratio $\log p_\theta(\mathbf{z})/q_\psi(\mathbf{z}|\mathbf{x})$, which is needed to compute the ELBO. However, we can use the same method as is used in GANs (generative adversarial networks, Chapter 26), in which we train a classifier that discriminates prior samples from samples from the variational posterior by evaluating $T(\mathbf{x}, \mathbf{z}) = \log q_\psi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{z})$. See e.g., [TR19] for details.

10.4.5 Combining VI with MCMC inference

There are various ways to combine variational inference with MCMC to get an improved approximate posterior. In [SKW15], they propose **Hamiltonian variational inference**, in which they train an inference network to initialize an HMC sampler (Section 12.5). The gradient of the log posterior (wrt the latents), which is needed by HMC, is given by

$$\nabla_{\mathbf{z}} \log p_\theta(\mathbf{z}|\mathbf{x}) = \nabla_{\mathbf{z}} \log [p_\theta(\mathbf{x}, \mathbf{z}) - \log p_\theta(\mathbf{x})] = \nabla_{\mathbf{z}} \log p_\theta(\mathbf{x}, \mathbf{z}) \quad (10.165)$$

This is easy to compute. They use the final sample to approximate the posterior $q_\phi(\mathbf{z}|\mathbf{x})$. To compute the entropy of this distribution, they also learn an auxiliary inverse inference network to reverse the HMC Markov chain.

A simpler approach is proposed in [Hof17]. Here they train an inference network to initialize an HMC sampler, using the standard ELBO for ϕ , but they optimize the generative parameters θ using

a stochastic approximation to the log marginal likelihood, given by $\log p_{\theta}(\mathbf{z}, \mathbf{x})$ where \mathbf{z} is a sample from the HMC chain. This does not require learning a reverse inference network, and avoids problems with variational pruning, since it does not use the ELBO for training the generative model.

10.5 Tighter bounds

Another way to improve the quality of the posterior approximation is to optimize q wrt a bound that is a tighter approximation to the log marginal likelihood compared to the standard ELBO. We give some examples below.

10.5.1 Multi-sample ELBO (IWAE bound)

In this section, we discuss a method known as the **importance weighted autoencoder** or **IWAE** [BGS16], which is a way to tighten the variational lower bound by using self-normalized importance sampling (Section 11.5.2). (It can also be interpreted as standard ELBO maximization in an expanded model, where we add extra auxiliary variables [CMD17; DS18; Tuc+19].)

Let the inference network $q_{\phi}(\mathbf{z}|\mathbf{x})$ be viewed as a proposal distribution for the target posterior $p_{\theta}(\mathbf{z}|\mathbf{x})$. Define $w_s^* = \frac{p_{\theta}(\mathbf{x}, \mathbf{z}_s)}{q_{\phi}(\mathbf{z}_s|\mathbf{x})}$ as the unnormalized importance weight for a sample, and $w_s = w_s^*/(\sum_{s'=1}^S w_{s'}^*)$ as the normalized importance weights. From Equation (11.43) we can compute an estimate of the marginal likelihood $p(\mathbf{x})$ using

$$\hat{p}_S(\mathbf{x}|\mathbf{z}_{1:S}) \triangleq \frac{1}{S} \sum_{k=1}^S \frac{p_{\theta}(\mathbf{x}, \mathbf{z}_s)}{q_{\phi}(\mathbf{z}_s|\mathbf{x})} = \frac{1}{S} \sum_{k=1}^S w_s \quad (10.166)$$

This is unbiased, i.e., $\mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\hat{p}_S(\mathbf{x}|\mathbf{z}_{1:S})] = p(\mathbf{x})$, where $q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x}) = \prod_{s=1}^S q_{\phi}(\mathbf{z}_s|\mathbf{x})$. In addition, since the estimator is always positive, we can take logarithms, and thus obtain a stochastic lower bound on the log likelihood:

$$\mathbb{L}_S(\phi, \theta | \mathbf{x}) \triangleq \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} \left[\log \left(\frac{1}{S} \sum_{s=1}^S w_s \right) \right] = \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\log \hat{p}_S(\mathbf{z}_{1:S})] \quad (10.167)$$

$$\leq \log \mathbb{E}_{q_{\phi}(\mathbf{z}_{1:S}|\mathbf{x})} [\hat{p}_S(\mathbf{z}_{1:S})] = \log p(\mathbf{x}) \quad (10.168)$$

where we used Jensen's inequality in the penultimate line, and the unbiased property in the last line. This is called the **multi-sample ELBO** or **IWAE bound** [BGS16]. The gradients of this expression wrt θ and ϕ are given in Equation (10.179). If $S = 1$, \mathbb{L}_S reduces to the standard ELBO:

$$\mathbb{L}_1(\phi, \theta | \mathbf{x}) = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} [\log w] = \int q_{\phi}(\mathbf{z}|\mathbf{x}) \log \frac{p_{\theta}(\mathbf{z}, \mathbf{x})}{q_{\phi}(\mathbf{z}|\mathbf{x})} d\mathbf{z} \quad (10.169)$$

One can show [BGS16] that increasing the number of samples S is guaranteed to make the bound tighter, thus making it a better proxy for the log likelihood. Intuitively, averaging the S samples inside the log removes the need for every sample \mathbf{z}_s to explain the data \mathbf{x} . This encourages the proposal distribution q to be less concentrated than the single-sample variational posterior.

10.5.1.1 Pathologies of optimizing the IWAE bound

Unfortunately, increasing the number of samples in the IWAE bound can decrease the signal to noise ratio, resulting in learning a worse model [Rai+18a]. Intuitively, the reason this happens is that increasing S reduces the dependence of the bound on the quality of the inference network, which makes the gradient of the ELBO wrt ϕ less informative (higher variance).

One solution to this is to use the **doubly reparameterized gradient estimator** [TL18b]. Another approach is to use alternative estimation methods that avoid ELBO maximization, such as using the thermodynamic variational objective (see Section 10.5.2) or the reweighted wake-sleep algorithm (see Section 10.6).

10.5.2 The thermodynamic variational objective (TVO)

In [MLW19; Bre+20b], they present the **thermodynamic variational objective** or **TVO**. This is an alternative to IWAE for creating tighter variational bounds, which has certain advantages, particularly for posteriors that are not reparameterizable (e.g., discrete latent variables). The framework also has close connections with the reweighted wake-sleep algorithm from Section 10.6, as we will see in Section 10.5.3.

The TVO technique uses **thermodynamic integration**, also called **path sampling**, which is a technique used in physics and phylogenetics to approximate intractable normalization constants of high dimensional distributions (see e.g., [GM98; LP06; FP08]). This is based on the insight that it is easier to calculate the ratio of two unknown constants than to calculate the constants themselves. This is similar to the idea behind annealed importance sampling (Section 11.5.4), but TI is deterministic. For details, see [MLW19; Bre+20b].

10.5.3 Minimizing the evidence upper bound

Recall that the evidence lower bound or ELBO is given by

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - D_{\text{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{z} | \mathbf{x})) \leq \log p_{\boldsymbol{\theta}}(\mathbf{x}) \quad (10.170)$$

By analogy, we can define the **evidence upper bound** or **EUBO** as follows:

$$\text{EUBO}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) + D_{\text{KL}}(p_{\boldsymbol{\theta}}(\mathbf{z} | \mathbf{x}) \| q_{\boldsymbol{\phi}}(\mathbf{z} | \mathbf{x})) \geq \log p_{\boldsymbol{\theta}}(\mathbf{x}) \quad (10.171)$$

Minimizing this wrt the variational parameters $\boldsymbol{\phi}$, as an alternative to maximizing the ELBO, was proposed in [MLW19], where they showed that it can sometimes converge to the true $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ faster.

The above bound is for a specific input \mathbf{x} . If we sample \mathbf{x} from the generative model, and minimize $\mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{x})} [\text{EUBO}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x})]$ wrt $\boldsymbol{\phi}$, we recover the sleep phase of the wake-sleep algorithm (see Section 10.6.2).

Now suppose we sample \mathbf{x} from the empirical distribution, and minimize $\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [\text{EUBO}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x})]$ wrt $\boldsymbol{\phi}$. To approximate the expectation, we can use self-normalized importance sampling, as in Equation (10.188), to get

$$\nabla_{\boldsymbol{\phi}} \text{EUBO}(\boldsymbol{\theta}, \boldsymbol{\phi} | \mathbf{x}) = \sum_{s=1}^S \bar{w}_s \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z}^s | \mathbf{x}) \quad (10.172)$$

where $\bar{w}_s = w^{(s)} / (\sum_{s'} w^{(s')})$, and $w^{(s)} = \frac{p(\mathbf{x}, \mathbf{z}^s)}{q(\mathbf{z}^s | \phi_t)}$. This is equivalent to the “daydream” update (aka “wake-phase ϕ update”) of the wake-sleep algorithm (see Section 10.6.3).

10.6 Wake-sleep algorithm

So far in this chapter we have focused on fitting latent variable models by maximizing the ELBO. This has two main drawbacks. First, it does not work well when we have discrete latent variables, because in such cases we cannot use the reparameterization trick; thus we have to use higher variance estimators, such as REINFORCE (see Section 10.2.3). Second, even in the case where we can use the reparameterization trick, the lower bound may not be very tight. We can improve the tightness by using the IWAE multi-sample bound (Section 10.5.1), but paradoxically this may not result in learning a better model, for reasons discussed in Section 10.5.1.1.

In this section, we discuss a different way to jointly train generative and inference models, which avoids some of the problems with ELBO maximization. The method is known as the **wake-sleep algorithm** [Hin+95; BB15b; Le+19; FT19], because it alternates between two steps: in the wake phase, we optimize the generative model parameters θ to maximize the marginal likelihood of the observed data (we approximate $\log p_\theta(\mathbf{x})$ by drawing importance samples from the inference network), and in the sleep phase, we optimize the inference model parameters ϕ to learn to invert the generative model by training the inference network on labeled (\mathbf{x}, \mathbf{z}) pairs, where \mathbf{x} are samples generated by the current model parameters. This can be viewed as a form of **adaptive importance sampling**, which iteratively improves its proposal, while simultaneously optimizing the model. We give further details below.

10.6.1 Wake phase

In the **wake phase**, we minimize the KL divergence from the empirical distribution to the model’s distribution:

$$\mathcal{L}(\theta) = D_{\text{KL}}(p_D(\mathbf{x}) \| p_\theta(\mathbf{x})) = \mathbb{E}_{p_D(\mathbf{x})}[-\log p_\theta(\mathbf{x})] + \text{const} \quad (10.173)$$

where $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{z}) p_\theta(\mathbf{x} | \mathbf{z}) d\mathbf{z}$. This is equivalent to maximizing the likelihood of the observed data:

$$\ell(\theta) = \mathbb{E}_{p_D(\mathbf{x})}[\log p_\theta(\mathbf{x})] \quad (10.174)$$

Since the log marginal likelihood $\log p_\theta(\mathbf{x})$ cannot be computed exactly, we will approximate it. In the original wake-sleep paper, they proposed to use the ELBO lower bound. In the **reweighted wake-sleep** (RWS) algorithm of [BB15b; Le+19], they propose to use the IWAE bound from Section 10.5.1 instead. In particular, if we draw S samples from the inference network, $\mathbf{z}_s \sim q_\phi(\mathbf{z} | \mathbf{x})$, we get the following estimator:

$$\ell(\theta | \phi, \mathbf{x}) = \log \left(\frac{1}{S} \sum_{s=1}^S w_s \right) \quad (10.175)$$

where $w_s = \frac{p_\theta(\mathbf{x}, \mathbf{z}_s)}{q_\phi(\mathbf{z}_s | \mathbf{x})}$. Note that this is the same as the IWAE bound in Equation (10.168).

We now discuss how to compute the gradient of this objective wrt $\boldsymbol{\theta}$ or $\boldsymbol{\phi}$. Using the log-derivative trick, we have that

$$\nabla \log w_s = \frac{1}{w_s} \nabla w_s \quad (10.176)$$

Hence

$$\nabla \ell(\boldsymbol{\theta}|\boldsymbol{\phi}, \mathbf{x}) = \frac{1}{\frac{1}{S} \sum_{s=1}^S w_s} \left(\frac{1}{S} \sum_{s=1}^S \nabla w_s \right) \quad (10.177)$$

$$= \frac{1}{\sum_{s=1}^S w_s} \left(\sum_{s=1}^S w_s \nabla \log w_s \right) \quad (10.178)$$

$$= \sum_{s=1}^S \bar{w}_s \nabla \log w_s \quad (10.179)$$

where $\bar{w}_s = w_s / (\sum_{s'=1}^S w_{s'})$.

In the case of the derivatives wrt $\boldsymbol{\theta}$, we have

$$\nabla_{\boldsymbol{\theta}} \log w_s = \frac{1}{w_s} \nabla_{\boldsymbol{\theta}} w_s = \frac{q_{\boldsymbol{\phi}}(\mathbf{z}_s|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s)} \nabla_{\boldsymbol{\theta}} \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s)}{q_{\boldsymbol{\phi}}(\mathbf{z}_s|\mathbf{x})} = \frac{1}{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s)} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s) = \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s) \quad (10.180)$$

and hence we get

$$\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}|\boldsymbol{\phi}, \mathbf{x}) \sum_{s=1}^S \bar{w}_s \nabla \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}_s) \quad (10.181)$$

10.6.2 Sleep phase

In the **sleep phase**, we try to minimize the KL divergence between the true posterior (under the current model) and the inference network's approximation to that posterior:

$$\mathcal{L}(\boldsymbol{\phi}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{x})} [D_{\text{KL}}(p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) \| q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}))] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})} [-\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] + \text{const} \quad (10.182)$$

Equivalently, we can maximize the following log likelihood objective:

$$\ell(\boldsymbol{\phi}|\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{z}, \mathbf{x}) \sim p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})} [\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})] \quad (10.183)$$

where $p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. We see that the sleep phase amounts to maximum likelihood training of the inference network based on samples from the generative model. These “fantasy samples”, created while the network “dreams”, can be easily generated using ancestral sampling (Section 4.2.5). If we use S such samples, the objective becomes

$$\ell(\boldsymbol{\phi}|\boldsymbol{\theta}) = \frac{1}{S} \sum_{s=1}^S \log q_{\boldsymbol{\phi}}(\mathbf{z}'_s | \mathbf{x}'_s) \quad (10.184)$$

where $(\mathbf{z}'_s, \mathbf{x}'_s) \sim p_{\theta}(\mathbf{z}, \mathbf{x})$. The gradient of this is given by

$$\nabla_{\phi} \ell(\phi | \theta) = \frac{1}{S} \sum_{s=1}^S \nabla_{\phi} \log q_{\phi}(\mathbf{z}'_s | \mathbf{x}'_s) \quad (10.185)$$

We do not require $q_{\phi}(\mathbf{z}' | \mathbf{x})$ to be reparameterizable, since the samples are drawn from a distribution that is independent of ϕ . This means it is easy to apply this method to models with discrete latent variables.

10.6.3 Daydream phase

The disadvantage of the sleep phase is that the inference network, $q_{\phi}(\mathbf{z} | \mathbf{x})$, is trying to follow a moving target, $p_{\theta}(\mathbf{z} | \mathbf{x})$. Furthermore, it is only being trained on synthetic data from the model, not on real data. The reweighted wake-sleep algorithm of [BB15b] proposed to learn the inference network by using real data from the empirical distribution, in addition to fantasy data. They call the case where you use real data the “**wake-phase q update**”, but we will call it the “**daydream phase**”, since, unlike sleeping, the system uses real data \mathbf{x} to update the inference model, instead of fantasies.¹ [Le+19] went further, and proposed to only use the wake and daydream phases, and to skip the sleep phase entirely.

In more detail, the new objective which we want to minimize becomes

$$\mathcal{L}(\phi | \theta) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} [D_{\text{KL}}(p_{\theta}(\mathbf{z} | \mathbf{x}) \| q_{\phi}(\mathbf{z} | \mathbf{x}))] \quad (10.186)$$

We can compute a single sample approximation to the negative of the above expression as follows:

$$\ell(\phi | \theta, \mathbf{x}) = \mathbb{E}_{p_{\theta}(\mathbf{z} | \mathbf{x})} [\log q_{\phi}(\mathbf{z} | \mathbf{x})] \quad (10.187)$$

where $\mathbf{x} \sim p_{\mathcal{D}}$. We can approximate this expectation using importance sampling, with q_{ϕ} as the proposal. This results in the following estimator of the gradient for each datapoint:

$$\nabla_{\phi} \ell(\phi | \theta, \mathbf{x}) = \int p_{\theta}(\mathbf{z} | \mathbf{x}) \nabla_{\phi} \log q_{\phi}(\mathbf{z} | \mathbf{x}) d\mathbf{z} \approx \sum_{s=1}^S \bar{w}_s \nabla_{\phi} \log q_{\phi}(\mathbf{z}_s | \mathbf{x}) \quad (10.188)$$

where $\mathbf{z}_s \sim q_{\phi}(\mathbf{z}_s | \mathbf{x})$ and \bar{w}_s are the normalized weights.

We see that Equation (10.188) is very similar to Equation (10.185). The key difference is that in the daydream phase, we sample from $(\mathbf{x}, \mathbf{z}_s) \sim p_{\mathcal{D}}(\mathbf{x})q_{\phi}(\mathbf{z} | \mathbf{x})$, where \mathbf{x} is a real datapoint, whereas in the sleep phase, we sample from $(\mathbf{x}'_s, \mathbf{z}'_s) \sim p_{\theta}(\mathbf{z}, \mathbf{x})$, where \mathbf{x}'_s is generated datapoint.

10.6.4 Summary of algorithm

We summarize the RWS algorithm in Algorithm 10.5. The disadvantage of the RWS algorithm is that it does not optimize a single well-defined objective, so it is not clear if the method will converge, in contrast to ELBO maximization. On the other hand, the method is fairly simple, since it consists of two alternating weighted maximum likelihood problems. It can also be shown to “sandwich” a

1. We thank Rif A. Saurous for suggesting this term.

Algorithm 10.5: One SGD update using wake-sleep algorithm.

- 1 Sample \mathbf{x}_n from dataset
 - 2 Draw S samples from inference network: $\mathbf{z}_s \sim q(\mathbf{z}|\mathbf{x}_n)$
 - 3 Compute unnormalized weights: $w_s = \frac{p(\mathbf{x}_n, \mathbf{z}_s)}{q(\mathbf{z}_s|\mathbf{x}_n)}$
 - 4 Compute normalized weights: $\bar{w}_s = \frac{w_s}{\sum_{s'=1}^S w_{s'}}$
 - 5 Optional: Compute estimate of log likelihood: $\log p(\mathbf{x}_n) = \log(\frac{1}{S} \sum_{s=1}^S w_s)$
 - 6 Wake phase: Update $\boldsymbol{\theta}$ using $\sum_{s=1}^S \bar{w}_s \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{z}_s, \mathbf{x}_n)$
 - 7 Daydream phase: Update $\boldsymbol{\phi}$ using $\sum_{s=1}^S \bar{w}_s \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z}_s|\mathbf{x}_n)$
 - 8 Optional sleep phase: Draw S samples from model, $(\mathbf{x}'_s, \mathbf{z}'_s) \sim p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ and update $\boldsymbol{\phi}$ using $\frac{1}{S} \sum_{s=1}^S \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z}'_s|\mathbf{x}'_s)$
 - 9 b
-

lower and upper bound of the log marginal likelihood. We can think of this in terms of the two joint distributions $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z})p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ and $q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) = p_{\mathcal{D}}(\mathbf{x})q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$:

$$\text{wake phase } \min_{\boldsymbol{\theta}} D_{\text{KL}}(q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})) \quad (10.189)$$

$$\text{daydream phase } \min_{\boldsymbol{\phi}} D_{\text{KL}}(p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) \parallel q_{\mathcal{D}, \boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})) \quad (10.190)$$

10.7 Expectation propagation (EP)

One problem with lower bound maximization (i.e., standard VI) is that we are minimizing $D_{\text{KL}}(q \parallel p)$, which induces **zero-forcing** behavior, as we discussed in Section 5.1.4.1. This means that $q(\mathbf{z}|\mathbf{x})$ tends to be too compact (over-confident), to avoid the situation in which $q(\mathbf{z}|\mathbf{x}) > 0$ but $p(\mathbf{z}|\mathbf{x}) = 0$, which would incur infinite KL penalty.

Although zero-forcing can be desirable behavior for some multi-modal posteriors (e.g., mixture models), it is not so reasonable for many unimodal posteriors (e.g., Bayesian logistic regression, or GPs with log-concave likelihoods). One way to avoid this problem is to minimize $D_{\text{KL}}(p \parallel q)$, which is zero-avoiding, as we discussed in Section 5.1.4.1. This tends to result in broad posteriors, which avoids overconfidence. In this section, we discuss **expectation propagation** or EP [Min01b], which can be seen as a local approximation to $D_{\text{KL}}(p \parallel q)$.

10.7.1 Algorithm

We assume the exact posterior can be written as follows:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z_p} \hat{p}(\boldsymbol{\theta}), \quad \hat{p}(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta}) \prod_{k=1}^K f_k(\boldsymbol{\theta}) \quad (10.191)$$

where $\hat{p}(\boldsymbol{\theta})$ is the unnormalized posterior, p_0 is the prior, f_k corresponds to the k 'th likelihood term or **local factor** (also called a **site potential**). Here $Z_p = p(\mathcal{D})Z_0$ is the normalization constant for

the posterior, where Z_0 is the normalization constant for the prior. To simplify notation, we let $f_0(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta})$ be the prior.

We will approximate the posterior as follows:

$$q(\boldsymbol{\theta}) = \frac{1}{Z_q} \hat{q}(\boldsymbol{\theta}), \quad \hat{q}(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta}) \prod_{k=1}^K \tilde{f}_k(\boldsymbol{\theta}) \quad (10.192)$$

where $\tilde{f}_k \in \mathcal{Q}$ is the approximate local factor, and \mathcal{Q} is some tractable family in the exponential family, usually a Gaussian [Gel+14b].

We will optimize each \tilde{f}_i in turn, keeping the others fixed. We initialize each \tilde{f}_i using an uninformative distribution from the family \mathcal{Q} , so $q(\boldsymbol{\theta}) = p_0(\boldsymbol{\theta})$.

To compute the new local factor \tilde{f}_i^{new} , we proceed as follows. First we compute the **cavity distribution** by deleting the \tilde{f}_i from the approximate posterior by dividing it out:

$$q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) = \frac{q(\boldsymbol{\theta})}{\tilde{f}_i(\boldsymbol{\theta})} \propto \prod_{k \neq i} \tilde{f}_k(\boldsymbol{\theta}) \quad (10.193)$$

This division operation can be implemented by subtracting the natural parameters, as explained in Section 2.3.3.2. The cavity distribution represents the effect of all the factors except for f_i (which is approximated by \tilde{f}_i).

Next we (conceptually) compute the **tilted distribution** by multiplying the exact factor f_i onto the cavity distribution:

$$q_i^{\text{tilted}}(\boldsymbol{\theta}) = \frac{1}{Z_i} f_i(\boldsymbol{\theta}) q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) \quad (10.194)$$

where $Z_i = \int q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) f_i(\boldsymbol{\theta}) d\boldsymbol{\theta}$ is the normalization constant for the tilted distribution. This is the result of combining the current approximation, excluding factor i , with the exact f_i term.

Unfortunately, the resulting tilted distribution may be outside of our model family (e.g., if we combine a Gaussian prior with a non-Gaussian likelihood). So we will approximate the tilted distribution as follows:

$$q_i^{\text{proj}}(\boldsymbol{\theta}) = \text{proj}(q_i^{\text{tilted}}) \triangleq \underset{\tilde{q} \in \mathcal{Q}}{\operatorname{argmin}} D(q_i^{\text{tilted}} || \tilde{q}) \quad (10.195)$$

This can be thought of as projecting the tilted distribution into the approximation family. If $D(q_i^{\text{tilted}} || q) = D_{\text{KL}}(q_i^{\text{tilted}} || q)$, this can be done by moment matching, as shown in Section 5.1.4.2. For example, suppose the cavity distribution is Gaussian, $q_{-i}^{\text{cavity}}(\boldsymbol{\theta}) = \mathcal{N}_c(\boldsymbol{\theta} | \mathbf{r}_{-i}, \mathbf{Q}_{-i})$, using the canonical parameterization. Then the log of the tilted distribution is given by

$$\log q_i^{\text{tilted}}(\boldsymbol{\theta}) = \alpha \log f_i(\boldsymbol{\theta}) - \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{Q}_{-i} \boldsymbol{\theta} + \mathbf{r}_{-i}^\top \boldsymbol{\theta} + \text{const} \quad (10.196)$$

Let $\hat{\boldsymbol{\theta}}$ be a local maximum of this objective. If \mathcal{Q} is the set of Gaussians, we can compute the projected tilted distribution as a Gaussian with the following parameters:

$$\mathbf{Q}_{\setminus i} = -\nabla_{\boldsymbol{\theta}}^2 \log q_i^{\text{tilted}}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}}, \quad \mathbf{r}_{\setminus i} = \mathbf{Q}_{\setminus i} \hat{\boldsymbol{\theta}} \quad (10.197)$$

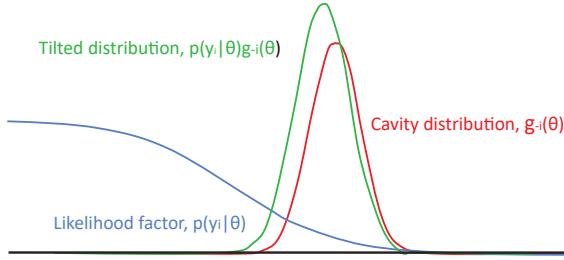


Figure 10.15: Combining a logistic likelihood factor $f_i = p(y_i | \theta)$ with the cavity prior, $q_{-i}^{cavity} = g_{-i}(\theta)$, to get the tilted distribution, $q_i^{tilted} = p(y_i | \theta)g_{-i}(\theta)$. Adapted from Figure 2 of [Gel+14b].

This is called **Laplace propagation** [SVE04]. For more general distributions, we can use Monte Carlo approximations; this is known as **blackbox EP** [HL+16a; Li+18c].

Finally, we compute a local factor that, if combined with the cavity distribution, would give the same results as this projected distribution:

$$\tilde{f}_i^{\text{new}}(\theta) = \frac{q_i^{\text{proj}}(\theta)}{q_{-i}^{\text{cavity}}(\theta)} \quad (10.198)$$

We see that $q_{-i}^{\text{cavity}}(\theta)\tilde{f}_i^{\text{new}}(\theta) = q_i^{\text{proj}}(\theta)$, so combining this approximate factor with the cavity distribution results in a distribution which is the best possible approximation (within \mathcal{Q}) to the results of using the exact factor.

10.7.2 Example

Figure 10.15 illustrates the process of combining a very non-Gaussian likelihood f_i with a Gaussian cavity prior q_{-i}^{cavity} to yield a nearly Gaussian tilted distribution q_i^{tilted} , which can then be approximated by a Gaussian using projection.

Thus instead of trying to “Gaussianize” each likelihood term f_i in isolation (as is done, e.g., in EKF), we try to find the best local factor \tilde{f}_i (within some family) that achieves approximately the same effect, when combined with all the other terms (represented by the cavity distribution, q_{-i}), as using the exact factor f_i . That is, we choose a local factor that works well in the context of all the other factors.

10.7.3 EP as generalized ADF

We can view EP as a generalization of the ADF algorithm discussed in Section 8.6. ADF is a form of sequential Bayesian inference. At each step, it maintains a tractable approximation to the posterior, $q_t(z) \in \mathcal{Q}$, updates it with the likelihood from the next observation, $\hat{p}_{t+1}(z) \propto q_t(z)p(x_t|z)$, and then projects the resulting updated posterior back to the tractable family using $q_{t+1} = \operatorname{argmin}_{q \in \mathcal{Q}} D_{\text{KL}}(\hat{p}_{t+1} \| q)$. ADF minimizes KL in the desired direction. However, it is a sequential algorithm, designed for the online setting. In the batch setting, the method can given

different results depending on the order in which the updates are performed. In addition, if we perform multiple passes over the data, we will include the same likelihood terms multiple times, resulting in an overconfident posterior. EP overcomes this problem.

10.7.4 Optimization issues

In practice, EP can be numerically unstable. For example, if we use Gaussians as our local factors, we might end up with negative variance when we subtract the natural parameters. To reduce the chance of this, it is common to use damping, in which we perform a partial update of each factor with a step size of δ . More precisely, we change the final step to be the following:

$$\tilde{f}_i^{\text{new}}(\boldsymbol{\theta}) = \left(\tilde{f}_i(\boldsymbol{\theta}) \right)^{1-\delta} \left(\frac{q_i^{\text{proj}}(\boldsymbol{\theta})}{q_{-i}^{\text{cavity}}} \right)^\delta \quad (10.199)$$

This can be implemented by scaling the natural parameters by δ . [ML02] suggest $\delta = 1/K$ as a safe strategy (where K is the number of factors), but this results in very slow convergence. [Gel+14b] suggest starting with $\delta = 0.5$, and then reducing to $\delta = 1/K$ over K iterations.

In addition to numerical stability, there is no guarantee that EP will converge in its vanilla form, although empirically it can work well, especially with log-concave factors f_i (e.g., as in GP classifiers).

10.7.5 Power EP and α -divergence

We also have a choice about what divergence measure $D(q_i^{\text{tilted}} \| q)$ to use when we approximate the tilted distribution. If we use $D_{\text{KL}}(q_i^{\text{tilted}} \| q)$, we recover classic EP, as described above. If we use $D_{\text{KL}}(q \| q_i^{\text{tilted}})$, we recover the reverse KL used in standard variational inference. We can generalize the above results by using α -divergences (Section 2.7.1.2), which allow us to interpolate between mode seeking and mode covering behavior, as shown in Figure 2.20. We can optimize the α -divergence by using the **power EP** method of [Min04].

Algorithmically, this is a fairly small modification to regular EP. In particular, we first compute the cavity distribution, $q_{-i}^{\text{cavity}} \propto \frac{q}{f_i^\alpha}$; we then approximate the tilted distribution, $q_i^{\text{proj}} = \text{proj}(q_{-i}^{\text{cavity}} f_i^\alpha)$; and finally we compute the new factor $\tilde{f}_i^{\text{new}} \propto \left(\frac{q_i^{\text{proj}}}{q_{-i}} \right)^{1/\alpha}$.

10.7.6 Stochastic EP

The main disadvantage of EP in the big data setting is that we need to store the $\tilde{f}_n(\boldsymbol{\theta})$ terms for each datapoint n , so we can compute the cavity distribution. If $\boldsymbol{\theta}$ has D dimensions, and we use full covariance Gaussians, this requires $O(ND^2)$ memory.

The idea behind **stochastic EP** [LHLT15] is to approximate the local factors with a shared factor that acts like an aggregated likelihood, i.e.,

$$\prod_{n=1}^N f_n(\boldsymbol{\theta}) \approx \tilde{f}(\boldsymbol{\theta})^N \quad (10.200)$$

where typically $f_n(\boldsymbol{\theta}) = p(\mathbf{x}_n | \boldsymbol{\theta})$. This exploits the fact that the posterior only cares about approximating the product of the likelihoods, rather than each likelihood separately. Hence it suffices for $\tilde{f}(\boldsymbol{\theta})$ to approximate the average likelihood.

We can modify EP to this setting as follows. First, when computing the cavity distribution, we use

$$q_{-1}(\boldsymbol{\theta}) \propto q(\boldsymbol{\theta}) / \tilde{f}(\boldsymbol{\theta}) \quad (10.201)$$

We then compute the tilted distribution

$$q_{\setminus n}(\boldsymbol{\theta}) \propto f_n(\boldsymbol{\theta}) q_{-1}(\boldsymbol{\theta}) \quad (10.202)$$

Next we derive the new local factor for this datapoint using moment matching:

$$\tilde{f}_n(\boldsymbol{\theta}) = \text{proj}(q_{\setminus n}(\boldsymbol{\theta})) / q_{-1}(\boldsymbol{\theta}) \quad (10.203)$$

Finally, we perform a damped update of the average likelihood $\tilde{f}(\boldsymbol{\theta})$ using this new local factor:

$$\tilde{f}_{\text{new}}(\boldsymbol{\theta}) = \tilde{f}_{\text{old}}(\boldsymbol{\theta})^{1-1/N} \tilde{f}_n(\boldsymbol{\theta})^{1/N} \quad (10.204)$$

The ADF algorithm is similar to SEP, in that we compute the tilted distribution $q_{\setminus t} \propto f_t q_{t-1}$ and then project it, without needing to keep the f_t factors. The difference is that instead of using the cavity distribution $q_{-1}(\boldsymbol{\theta})$ as a prior, it uses the posterior from the previous time step, q_{t-1} . This avoids the need to compute and store \tilde{f} , but results in overconfidence in the batch setting.

11 Monte Carlo methods

11.1 Introduction

In this chapter, we discuss **Monte Carlo methods**, which are a stochastic approach to solving numerical integration problems. The name refers to the “Monte Carlo” casino in Monaco; this was used as a codename by von Neumann and Ulam, who invented the technique while working on the atomic bomb during WWII. Since then, the technique has become widely adopted in physics, statistics, machine learning, and many areas of science and engineering.

In this chapter, we give a brief introduction to some key concepts. In Chapter 12, we discuss MCMC, which is the most widely used MC method for high-dimensional problems. In Chapter 13, we discuss SMC, which is widely used for MC inference in state space models, but can also be applied more generally. For more details on MC methods, see e.g., [Liu01; RC04; KTB11; BZ20].

11.2 Monte Carlo integration

We often want to compute the expected value of some function of a random variable, $\mathbb{E}[f(\mathbf{X})]$. This requires computing the following integral:

$$\mathbb{E}[f(\mathbf{x})] = \int f(\mathbf{x})p(\mathbf{x})d\mathbf{x} \quad (11.1)$$

where $\mathbf{x} \in \mathbb{R}^n$, $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and $p(\mathbf{x})$ is the target distribution of \mathbf{X} .¹ In low dimensions (up to, say, 3), we can compute the above integral efficiently using **numerical integration**, which (adaptively) computes a grid, and then evaluates the function at each point on the grid.² But this does not scale to higher dimensions.

An alternative approach is to draw multiple random samples, $\mathbf{x}_n \sim p(\mathbf{x})$, and then to compute

$$\mathbb{E}[f(\mathbf{x})] \approx \frac{1}{N_s} \sum_{n=1}^{N_s} f(\mathbf{x}_n) \quad (11.2)$$

This is called **Monte Carlo integration**. It has the advantage over numerical integration that the function is only evaluated in places where there is non-negligible probability, so it does not

1. In many cases, the target distribution may be the posterior $p(\mathbf{x}|\mathbf{y})$, which can be hard to compute; in such problems, we often work with the unnormalized distribution, $\tilde{p}(\mathbf{x}) = p(\mathbf{x}, \mathbf{y})$, instead, and then normalize the results using $Z = \int p(\mathbf{x}, \mathbf{y})d\mathbf{x} = p(\mathbf{y})$.

2. In 1d, numerical integration is called **quadrature**; in higher dimensions, it is called **cubature** [Sar13].

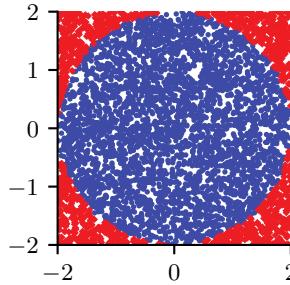


Figure 11.1: Estimating π by Monte Carlo integration using 5000 samples. Blue points are inside the circle, red points are outside. Generated by [mc_estimate_pi.ipynb](#).

need to uniformly cover the entire space. In particular, it can be shown that the accuracy is in principle independent of the dimensionality of \mathbf{x} , and only depends on the number of samples N_s (see Section 11.2.2 for details). The catch is that we need a way to generate the samples $\mathbf{x}_n \sim p(\mathbf{x})$ in the first place. In addition, the estimator may have high variance. We will discuss this topic at length in the sections below.

11.2.1 Example: estimating π by Monte Carlo integration

MC integration can be used for many applications, not just in ML and statistics. For example, suppose we want to estimate π . We know that the area of a circle with radius r is πr^2 , but it is also equal to the following definite integral:

$$I = \int_{-r}^r \int_{-r}^r \mathbb{I}(x^2 + y^2 \leq r^2) dx dy \quad (11.3)$$

Hence $\pi = I/(r^2)$. Let us approximate this by Monte Carlo integration. Let $f(x, y) = \mathbb{I}(x^2 + y^2 \leq r^2)$ be an indicator function that is 1 for points inside the circle, and 0 outside, and let $p(x)$ and $p(y)$ be uniform distributions on $[-r, r]$, so $p(x) = p(y) = 1/(2r)$. Then

$$I = (2r)(2r) \int \int f(x, y)p(x)p(y)dx dy \quad (11.4)$$

$$= 4r^2 \int \int f(x, y)p(x)p(y)dx dy \quad (11.5)$$

$$\approx 4r^2 \frac{1}{N_s} \sum_{n=1}^{N_s} f(x_n, y_n) \quad (11.6)$$

Using 5000 samples, we find $\hat{\pi} = 3.10$ with standard error 0.09 compared to the true value of $\pi = 3.14$. We can plot the points that are accepted or rejected as in Figure 11.1.

11.2.2 Accuracy of Monte Carlo integration

The accuracy of an MC approximation increases with sample size. This is illustrated in Figure 11.2. On the top line, we plot a histogram of samples from a Gaussian distribution. On the bottom line,

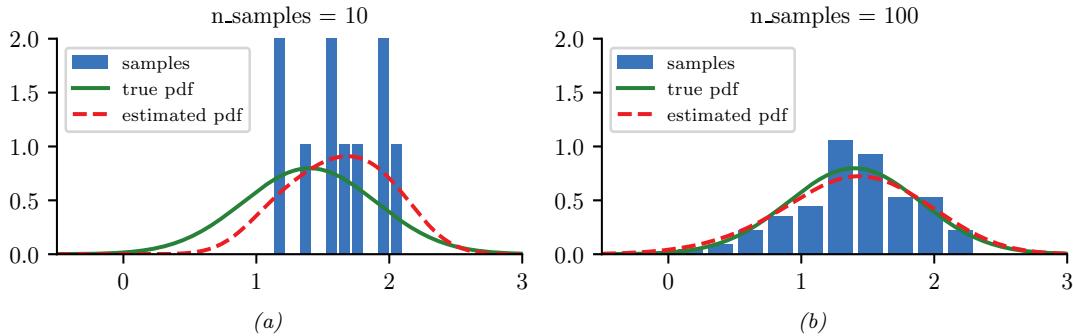


Figure 11.2: 10 and 100 samples from a Gaussian distribution, $\mathcal{N}(\mu = 1.5, \sigma^2 = 0.25)$. A dotted red line denotes kernel density estimate derived from the samples. Generated by [mc_accuracy_demo.ipynb](#).

we plot a smoothed version of these samples, created using a kernel density estimate. This smoothed distribution is then evaluated on a dense grid of points and plotted. Note that this smoothing is just for the purposes of plotting, it is not used for the Monte Carlo estimate itself.

If we denote the exact mean by $\mu = \mathbb{E}[f(X)]$, and the MC approximation by $\hat{\mu}$, one can show that, with independent samples,

$$(\hat{\mu} - \mu) \rightarrow \mathcal{N}(0, \frac{\sigma^2}{N_s}) \quad (11.7)$$

where

$$\sigma^2 = \mathbb{V}[f(X)] = \mathbb{E}[f(X)^2] - \mathbb{E}[f(X)]^2 \quad (11.8)$$

This is a consequence of the central limit theorem. Of course, σ^2 is unknown in the above expression, but it can be estimated by MC:

$$\hat{\sigma}^2 = \frac{1}{N_s} \sum_{n=1}^{N_s} (f(x_n) - \hat{\mu})^2 \quad (11.9)$$

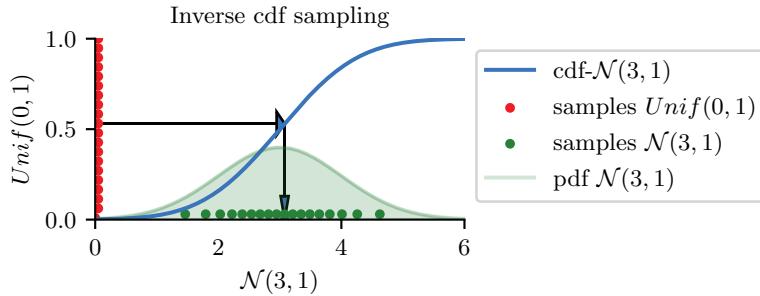
Thus for large enough N_s we have

$$P\left\{ \hat{\mu} - 1.96 \frac{\hat{\sigma}}{\sqrt{N_s}} \leq \mu \leq \hat{\mu} + 1.96 \frac{\hat{\sigma}}{\sqrt{N_s}} \right\} \approx 0.95 \quad (11.10)$$

The term $\sqrt{\hat{\sigma}^2/N_s}$ is called the (numerical or empirical) **standard error**, and is an estimate of our uncertainty about our estimate of μ .

If we want to report an answer which is accurate to within $\pm\epsilon$ with probability at least 95%, we need to use a number of samples N_s which satisfies $1.96\sqrt{\hat{\sigma}^2/N_s} \leq \epsilon$. We can approximate the 1.96 factor by 2, yielding $N_s \geq \frac{4\hat{\sigma}^2}{\epsilon^2}$.

The remarkable thing to note about the above results is that the error in the estimate, σ^2/N_s , is theoretically independent of the dimensionality of the integral. The catch is that sampling from high dimensional distributions can be hard. We turn to that topic next.

Figure 11.3: Sampling from $\mathcal{N}(3, 1)$ using an inverse cdf.

11.3 Generating random samples from simple distributions

We saw in Section 11.2 how we can evaluate $\mathbb{E}[f(X)]$ for different functions f of a random variable X using Monte Carlo integration. The main computational challenge is to efficiently generate samples from the probability distribution $p^*(\mathbf{x})$ (which may be a posterior, $p^*(\mathbf{x}) \propto p(\mathbf{x}|\mathcal{D})$). In this section, we discuss sampling methods that are suitable for parametric univariate distributions. These can be used as building blocks for sampling from more complex multivariate distributions.

11.3.1 Sampling using the inverse cdf

The simplest method for sampling from a univariate distribution is based on the **inverse probability transform**. Let F be a cdf of some distribution we want to sample from, and let F^{-1} be its inverse. Then we have the following result.

Theorem 11.3.1. *If $U \sim U(0, 1)$ is a uniform rv, then $F^{-1}(U) \sim F$.*

Proof.

$$\Pr(F^{-1}(U) \leq x) = \Pr(U \leq F(x)) \quad (\text{applying } F \text{ to both sides}) \quad (11.11)$$

$$= F(x) \quad (\text{because } \Pr(U \leq y) = y) \quad (11.12)$$

where the first line follows since F is a monotonic function, and the second line follows since U is uniform on the unit interval. \square

Hence we can sample from any univariate distribution, for which we can evaluate its inverse cdf, as follows: generate a random number $u \sim U(0, 1)$ using a **pseudorandom number generator** (see e.g., [Pre+88] for details). Let u represent the height up the y axis. Then “slide along” the x axis until you intersect the F curve, and then “drop down” and return the corresponding x value. This corresponds to computing $x = F^{-1}(u)$. See Figure 11.3 for an illustration.

For example, consider the exponential distribution

$$\text{Expon}(x|\lambda) \triangleq \lambda e^{-\lambda x} \mathbb{I}(x \geq 0) \quad (11.13)$$

The cdf is

$$F(x) = 1 - e^{-\lambda x} \mathbb{I}(x \geq 0) \quad (11.14)$$

whose inverse is the quantile function

$$F^{-1}(p) = -\frac{\ln(1-p)}{\lambda} \quad (11.15)$$

By the above theorem, if $U \sim \text{Unif}(0, 1)$, we know that $F^{-1}(U) \sim \text{Expon}(\lambda)$. So we can sample from the exponential distribution by first sampling from the uniform and then transforming the results using $-\ln(1-u)/\lambda$. (In fact, since $1-U \sim \text{Unif}(0, 1)$, we can just use $-\ln(u)/\lambda$.)

11.3.2 Sampling from a Gaussian (Box-Muller method)

In this section, we describe a method to sample from a Gaussian. The idea is we sample uniformly from a unit radius circle, and then use the change of variables formula to derive samples from a spherical 2d Gaussian. This can be thought of as two samples from a 1d Gaussian.

In more detail, sample $z_1, z_2 \in (-1, 1)$ uniformly, and then discard pairs that do not satisfy $z_1^2 + z_2^2 \leq 1$. The result will be points uniformly distributed inside the unit circle, so $p(\mathbf{z}) = \frac{1}{\pi} \mathbb{I}(z \text{ inside circle})$. Now define

$$x_i = z_i \left(\frac{-2 \ln r^2}{r^2} \right)^{\frac{1}{2}} \quad (11.16)$$

for $i = 1 : 2$, where $r^2 = z_1^2 + z_2^2$. Using the multivariate change of variables formula, we have

$$p(x_1, x_2) = p(z_1, z_2) \left| \frac{\partial(z_1, z_2)}{\partial(x_1, x_2)} \right| = \left[\frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x_1^2) \right] \left[\frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x_2^2) \right] \quad (11.17)$$

Hence x_1 and x_2 are two independent samples from a univariate Gaussian. This is known as the **Box-Muller** method.

To sample from a multivariate Gaussian, we first compute the Cholesky decomposition of its covariance matrix, $\Sigma = \mathbf{L}\mathbf{L}^\top$, where \mathbf{L} is lower triangular. Next we sample $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ using the Box-Muller method. Finally we set $\mathbf{y} = \mathbf{L}\mathbf{x} + \boldsymbol{\mu}$. This is valid since

$$\text{Cov}[\mathbf{y}] = \mathbf{LCov}[\mathbf{x}] \mathbf{L}^\top = \mathbf{L} \mathbf{I} \mathbf{L}^\top = \Sigma \quad (11.18)$$

11.4 Rejection sampling

Suppose we want to sample from the **target distribution**

$$p(\mathbf{x}) = \tilde{p}(\mathbf{x}) / Z_p \quad (11.19)$$

where $\tilde{p}(\mathbf{x})$ is the unnormalized version, and

$$Z_p = \int \tilde{p}(\mathbf{x}) d\mathbf{x} \quad (11.20)$$

is the (possibly unknown) normalization constant. One of the simplest approaches to this problem is **rejection sampling**, which we now explain.

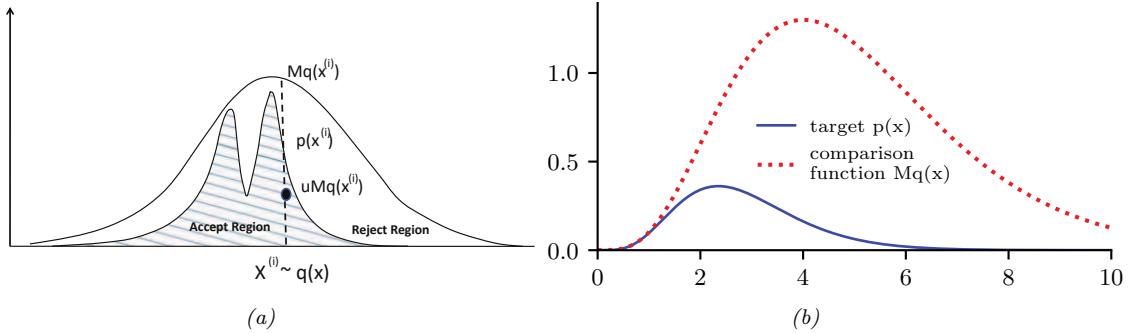


Figure 11.4: (a) Schematic illustration of rejection sampling. From Figure 2 of [And+03]. Used with kind permission of Nando de Freitas. (b) Rejection sampling from a $\text{Ga}(\alpha = 5.7, \lambda = 2)$ distribution (solid blue) using a proposal of the form $M\text{Ga}(k, \lambda - 1)$ (dotted red), where $k = \lfloor 5.7 \rfloor = 5$. The curves touch at $\alpha - k = 0.7$. Generated by [rejection_sampling_demo.ipynb](#).

11.4.1 Basic idea

In rejection sampling, we require access to a **proposal distribution** $q(\mathbf{x})$ which satisfies $Cq(\mathbf{x}) \geq \tilde{p}(\mathbf{x})$, for some constant C . The function $Cq(\mathbf{x})$ provides an upper envelope for \tilde{p} .

We can use the proposal distribution to generate samples from the target distribution as follows. We first sample $\mathbf{x}_0 \sim q(\mathbf{x})$, which corresponds to picking a random \mathbf{x} location, and then we sample $u_0 \sim \text{Unif}(0, Cq(\mathbf{x}_0))$, which corresponds to picking a random height (y location) under the envelope. If $u_0 > \tilde{p}(\mathbf{x}_0)$, we reject the sample, otherwise we accept it. This process is illustrated in 1d in Figure 11.4(a): the acceptance region is shown shaded, and the rejection region is the white region between the shaded zone and the upper envelope.

We now prove this procedure is correct. First note that the probability of any given sample \mathbf{x}_0 being accepted equals the probability of a sample $u_0 \sim \text{Unif}(0, Cq(\mathbf{x}_0))$ being less than or equal to $\tilde{p}(\mathbf{x}_0)$, i.e.,

$$q(\text{accept} | \mathbf{x}_0) = \int_0^{\tilde{p}(\mathbf{x}_0)} \frac{1}{Cq(\mathbf{x}_0)} du = \frac{\tilde{p}(\mathbf{x}_0)}{Cq(\mathbf{x}_0)} \quad (11.21)$$

Therefore

$$q(\text{propose and accept } \mathbf{x}_0) = q(\mathbf{x}_0)q(\text{accept} | \mathbf{x}_0) = q(\mathbf{x}_0) \frac{\tilde{p}(\mathbf{x}_0)}{Cq(\mathbf{x}_0)} = \frac{\tilde{p}(\mathbf{x}_0)}{C} \quad (11.22)$$

Integrating both sides give

$$\int q(\mathbf{x}_0)q(\text{accept} | \mathbf{x}_0) d\mathbf{x}_0 = q(\text{accept}) = \frac{\int \tilde{p}(\mathbf{x}_0) d\mathbf{x}_0}{C} = \frac{Z_p}{C} \quad (11.23)$$

Hence we see that the distribution of accepted points is given by the target distribution:

$$q(\mathbf{x}_0 | \text{accept}) = \frac{q(\mathbf{x}_0, \text{accept})}{q(\text{accept})} = \frac{\tilde{p}(\mathbf{x}_0)}{C} \frac{C}{Z_p} = \frac{\tilde{p}(\mathbf{x}_0)}{Z_p} = p(\mathbf{x}_0) \quad (11.24)$$

How efficient is this method? If \tilde{p} is a normalized target distribution, the acceptance probability is $1/C$. Hence we want to choose C as small as possible while still satisfying $Cq(x) \geq \tilde{p}(x)$.

11.4.2 Example

For example, suppose we want to sample from a gamma distribution:³

$$\text{Ga}(x|\alpha, \lambda) = \frac{1}{\Gamma(\alpha)} x^{\alpha-1} \lambda^\alpha \exp(-\lambda x) \quad (11.25)$$

where $\Gamma(\alpha)$ is the gamma function. One can show that if $X_i \stackrel{iid}{\sim} \text{Expon}(\lambda)$, and $Y = X_1 + \dots + X_k$, then $Y \sim \text{Ga}(k, \lambda)$. For non-integer shape parameters α , we cannot use this trick. However, we can use rejection sampling using a $\text{Ga}(k, \lambda - 1)$ distribution as a proposal, where $k = \lfloor \alpha \rfloor$. The ratio has the form

$$\frac{p(x)}{q(x)} = \frac{\text{Ga}(x|\alpha, \lambda)}{\text{Ga}(x|k, \lambda - 1)} = \frac{x^{\alpha-1} \lambda^\alpha \exp(-\lambda x) / \Gamma(\alpha)}{x^{k-1} (\lambda - 1)^k \exp(-(\lambda - 1)x) / \Gamma(k)} \quad (11.26)$$

$$= \frac{\Gamma(k) \lambda^\alpha}{\Gamma(\alpha) (\lambda - 1)^k} x^{\alpha-k} \exp(-x) \quad (11.27)$$

This ratio attains its maximum when $x = \alpha - k$. Hence

$$C = \frac{\text{Ga}(\alpha - k|\alpha, \lambda)}{\text{Ga}(\alpha - k|k, \lambda - 1)} \quad (11.28)$$

See Figure 11.4(b) for a plot.

11.4.3 Adaptive rejection sampling

We now describe a method that can automatically come up with a tight upper envelope $q(x)$ to any log concave 1d density $p(x)$. The idea is to upper bound the log density with a piecewise linear function, as illustrated in Figure 11.5(a). We choose the initial locations for the pieces based on a fixed grid over the support of the distribution. We then evaluate the gradient of the log density at these locations, and make the lines be tangent at these points.

Since the log of the envelope is piecewise linear, the envelope itself is piecewise exponential:

$$q(x) = C_i \lambda_i \exp(-\lambda_i(x - x_{i-1})), \quad x_{i-1} < x \leq x_i \quad (11.29)$$

where x_i are the grid points. It is relatively straightforward to sample from this distribution. If the sample x is rejected, we create a new grid point at x , and thereby refine the envelope. As the number of grid points is increased, the tightness of the envelope improves, and the rejection rate goes down. This is known as **adaptive rejection sampling** (ARS) [GW92]. Figure 11.5(b-c) gives an example of the method in action. As with standard rejection sampling, it can be applied to unnormalized distributions.

3. This section is based on notes by Ioana A. Cosma, available at <http://users.aims.ac.za/~ioana/cp2.pdf>.

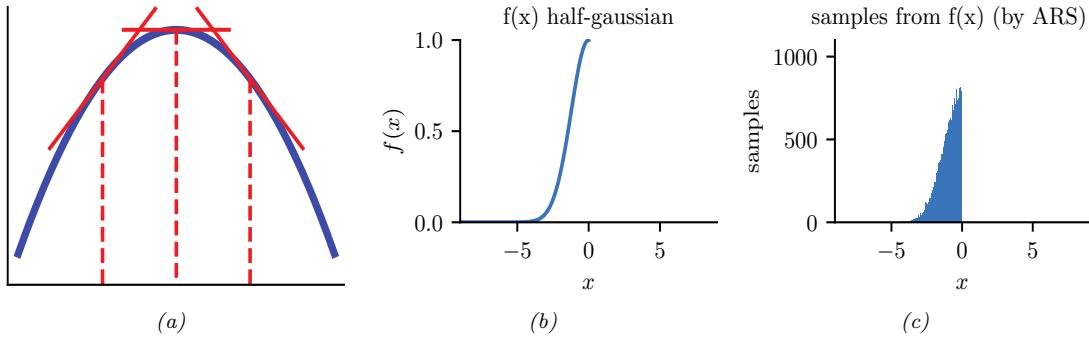


Figure 11.5: (a) Idea behind adaptive rejection sampling. We place piecewise linear upper (and lower) bounds on the log-concave density. Adapted from Figure 1 of [GW92]. Generated by `ars_envelope.ipynb`. (b-c) Using ARS to sample from a half-Gaussian. Generated by `ars_demo.ipynb`.

11.4.4 Rejection sampling in high dimensions

It is clear that we want to make our proposal $q(\mathbf{x})$ as close as possible to the target distribution $p(\mathbf{x})$, while still being an upper bound. But this is quite hard to achieve, especially in high dimensions. To see this, consider sampling from $p(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \sigma_p^2 \mathbf{I})$ using as a proposal $q(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \sigma_q^2 \mathbf{I})$. Obviously we must have $\sigma_q^2 \geq \sigma_p^2$ in order to be an upper bound. In D dimensions, the optimum value is given by $C = (\sigma_q/\sigma_p)^D$. The acceptance rate is $1/C$ (since both p and q are normalized), which decreases exponentially fast with dimension. For example, if σ_q exceeds σ_p by just 1%, then in 1000 dimensions the acceptance ratio will be about $1/20,000$. This is a fundamental weakness of rejection sampling.

11.5 Importance sampling

In this section, we describe a Monte Carlo method known as **importance sampling** for approximating integrals of the form

$$\mathbb{E} [\varphi(\mathbf{x})] = \int \varphi(\mathbf{x}) \pi(\mathbf{x}) d\mathbf{x} \quad (11.30)$$

where φ is called a **target function**, and $\pi(\mathbf{x})$ is the **target distribution**, often a conditional distribution of the form $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$. Since in general it is difficult to draw from the target distribution, we will instead draw from some **proposal distribution** $q(\mathbf{x})$ (which will usually depend on \mathbf{y}). We then adjust for the inaccuracies of this by associating weights with each sample, so we end up with a weighted MC approximation:

$$\mathbb{E} [\varphi(\mathbf{x})] \approx \sum_{n=1}^N W_n \varphi(\mathbf{x}_n) \quad (11.31)$$

We discuss two cases, first when the target is normalized, and then when it is unnormalized. This will affect the ways the weights are computed, as well as statistical properties of the estimator.

11.5.1 Direct importance sampling

In this section, we assume that we can *evaluate* the normalized target distribution $\pi(\mathbf{x})$, but we cannot sample from it. So instead we will sample from the proposal $q(\mathbf{x})$. We can then write

$$\int \varphi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \int \varphi(\mathbf{x})\frac{\pi(\mathbf{x})}{q(\mathbf{x})}q(\mathbf{x})d\mathbf{x} \quad (11.32)$$

We require that the proposal be non-zero whenever the target is non-zero, i.e., the support of $q(\mathbf{x})$ needs to be greater or equal to the support of $\pi(\mathbf{x})$. If we draw N_s samples $\mathbf{x}_n \sim q(\mathbf{x})$, we can write

$$\mathbb{E}[\varphi(\mathbf{x})] \approx \frac{1}{N_s} \sum_{n=1}^{N_s} \frac{\pi(\mathbf{x}_n)}{q(\mathbf{x}_n)} \varphi(\mathbf{x}_n) = \frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \varphi(\mathbf{x}_n) \quad (11.33)$$

where we have defined the **importance weights** as follows:

$$\tilde{w}_n = \frac{\pi(\mathbf{x}_n)}{q(\mathbf{x}_n)} \quad (11.34)$$

The result is an unbiased estimate of the true mean $\mathbb{E}[\varphi(\mathbf{x})]$.

11.5.2 Self-normalized importance sampling

The disadvantage of direct importance sampling is that we need a way to evaluate the normalized target distribution π in order to compute the weights. It is often much easier to evaluate the **unnormalized target distribution**

$$\tilde{\gamma}(\mathbf{x}) = Z\pi(\mathbf{x}) \quad (11.35)$$

where

$$Z = \int \tilde{\gamma}(\mathbf{x})d\mathbf{x} \quad (11.36)$$

is the normalization constant. (For example, if $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, then $\tilde{\gamma}(\mathbf{x}) = p(\mathbf{x}, \mathbf{y})$ and $Z = p(\mathbf{y})$.) The key idea is to also approximate the normalization constant Z with importance sampling. This method is called **self-normalized importance sampling**. The resulting estimate is a ratio of two estimates, and hence is biased. However as $N_s \rightarrow \infty$, the bias goes to zero, under some weak assumptions (see e.g., [RC04] for details).

In more detail, SNIS is based on this approximation:

$$\mathbb{E}[\varphi(\mathbf{x})] = \int \varphi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \frac{\int \varphi(\mathbf{x})\tilde{\gamma}(\mathbf{x})d\mathbf{x}}{\int \tilde{\gamma}(\mathbf{x})d\mathbf{x}} = \frac{\int \left[\frac{\tilde{\gamma}(\mathbf{x})}{q(\mathbf{x})} \varphi(\mathbf{x}) \right] q(\mathbf{x})d\mathbf{x}}{\int \left[\frac{\tilde{\gamma}(\mathbf{x})}{q(\mathbf{x})} \right] q(\mathbf{x})d\mathbf{x}} \quad (11.37)$$

$$\approx \frac{\frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \varphi(\mathbf{x}_n)}{\frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n} \quad (11.38)$$

where we have defined the **unnormalized weights**

$$\tilde{w}_n = \frac{\tilde{\gamma}(\mathbf{x}_n)}{q(\mathbf{x}_n)} \quad (11.39)$$

We can write Equation (11.38) more compactly as

$$\mathbb{E}[\varphi(\mathbf{x})] \approx \sum_{n=1}^{N_s} W_n \varphi(\mathbf{x}_n) \quad (11.40)$$

where we have defined the **normalized weights** by

$$W_n = \frac{\tilde{w}_n}{\sum_{n'=1}^{N_s} \tilde{w}_{n'}} \quad (11.41)$$

This is equivalent to approximating the target distribution using a weighted sum of delta functions:

$$\pi(\mathbf{x}) \approx \sum_{n=1}^{N_s} W_n \delta(\mathbf{x} - \mathbf{x}_n) \triangleq \hat{\pi}(\mathbf{x}) \quad (11.42)$$

As a byproduct of this algorithm we get the following approximation to the normalization constant:

$$Z \approx \frac{1}{N_s} \sum_{n=1}^{N_s} \tilde{w}_n \triangleq \hat{Z} \quad (11.43)$$

11.5.3 Choosing the proposal

The performance of importance sampling depends crucially on the quality of the proposal distribution. As we mentioned, we require that the support of q cover the support of the target (i.e., $\tilde{\gamma}(\mathbf{x}) > 0 \implies q(\mathbf{x}) > 0$). However, we also want the proposal to not be too “loose” or a “covering”. Ideally it should also take into account properties of the target function φ as well, as shown in Figure 11.6. This can yield substantial benefits, as shown in the “**target aware Bayesian inference**” scheme of [Rai+20]. However, usually the target function φ is unknown or ignored, so we just try to find a “generally useful” approximation to the target.

One way to come up with a good proposal is to learn one, by optimizing the variational lower bound or ELBO (see Section 10.1.1.2). Indeed, if we fix the parameters of the generative model, we can think of importance weighted autoencoders (Section 10.5.1) as learning a good IS proposal. More details on this connection can be found in [DS18].

11.5.4 Annealed importance sampling (AIS)

In this section, we describe a method known as **annealed importance sampling** [Nea01] for sampling from complex, possibly multimodal distributions. Assume we want to sample from some target distribution $p_0(\mathbf{x}) \propto f_0(\mathbf{x})$ (where $f_0(\mathbf{x})$ is the unnormalized version), but we cannot easily do so, because p_0 is complicated in some way (e.g., high dimensional and/or multi-modal). However, suppose that there is an easier distribution which we *can* sample from, call it $p_n(\mathbf{x}) \propto f_n(\mathbf{x})$; for

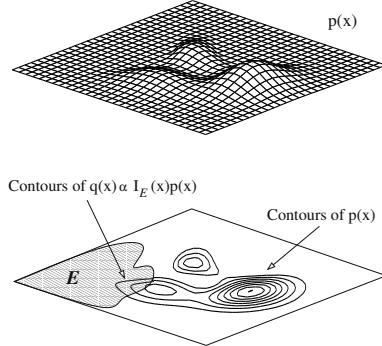


Figure 11.6: In importance sampling, we should sample from a distribution that takes into account regions where $\pi(\mathbf{x})$ has high probability and where $\varphi(\mathbf{x})$ is large. Here the function to be evaluated is an indicator function of a set, corresponding to a set of rare events in the tail of the distribution. From Figure 3 of [And+03]. Used with kind permission of Nando de Freitas.

example, this might be the prior. We now construct a sequence of intermediate distributions than move slowly from p_n to p_0 as follows:

$$f_j(\mathbf{x}) = f_0(\mathbf{x})^{\beta_j} f_n(\mathbf{x})^{1-\beta_j} \quad (11.44)$$

where $1 = \beta_0 > \beta_1 > \dots > \beta_n = 0$, where β_j is an inverse temperature. We will sample a set of points from f_n , and then from f_{n-1} , and so on, until we eventually sample from f_0 .

To sample from each f_j , suppose we can define a Markov chain $T_j(\mathbf{x}, \mathbf{x}') = p_j(\mathbf{x}'|\mathbf{x})$, which leaves p_0 invariant (i.e., $\int p_j(\mathbf{x}'|\mathbf{x})p_0(\mathbf{x})d\mathbf{x} = p_0(\mathbf{x}')$). (See Chapter 12 for details on how to construct such chains.) Given this, we can sample \mathbf{x} from p_0 as follows: sample $\mathbf{v}_n \sim p_n$; sample $\mathbf{v}_{n-1} \sim T_{n-1}(\mathbf{v}_n, \cdot)$; and continue in this way until we sample $\mathbf{v}_0 \sim T_0(\mathbf{v}_1, \cdot)$; finally we set $\mathbf{x} = \mathbf{v}_0$ and give it weight

$$w = \frac{f_{n-1}(\mathbf{v}_{n-1})}{f_n(\mathbf{v}_{n-1})} \frac{f_{n-2}(\mathbf{v}_{n-2})}{f_{n-1}(\mathbf{v}_{n-2})} \dots \frac{f_1(\mathbf{v}_1)}{f_2(\mathbf{v}_1)} \frac{f_0(\mathbf{v}_0)}{f_1(\mathbf{v}_0)} \quad (11.45)$$

This can be shown to be correct by viewing the algorithm as a form of importance sampling in an extended state space $\mathbf{v} = (\mathbf{v}_0, \dots, \mathbf{v}_n)$. Consider the following distribution on this state space:

$$p(\mathbf{v}) \propto \varphi(\mathbf{v}) = f_0(\mathbf{v}_0)\tilde{T}_0(\mathbf{v}_0, \mathbf{v}_1)\tilde{T}_2(\mathbf{v}_1, \mathbf{v}_2) \dots \tilde{T}_{n-1}(\mathbf{v}_{n-1}, \mathbf{v}_n) \quad (11.46)$$

$$\propto p(\mathbf{v}_0)p(\mathbf{v}_1|\mathbf{v}_0) \dots p(\mathbf{v}_n|\mathbf{v}_{n-1}) \quad (11.47)$$

where \tilde{T}_j is the reversal of T_j :

$$\tilde{T}_j(\mathbf{v}, \mathbf{v}') = T_j(\mathbf{v}', \mathbf{v})p_j(\mathbf{v}')/p_j(\mathbf{v}) = T_j(\mathbf{v}', \mathbf{v})f_j(\mathbf{v}')/f_j(\mathbf{v}) \quad (11.48)$$

It is clear that $\sum_{\mathbf{v}_1, \dots, \mathbf{v}_n} \varphi(\mathbf{v}) = f_0(\mathbf{v}_0)$, so by sampling from $p(\mathbf{v})$, we can effectively sample from $p_0(\mathbf{x})$.

We can sample on this extended state space using the above algorithm, which corresponds to the following proposal:

$$q(\mathbf{v}) \propto g(\mathbf{v}) = f_n(\mathbf{v}_n)T_{n-1}(\mathbf{v}_n, \mathbf{v}_{n-1}) \cdots T_2(\mathbf{v}_2, \mathbf{v}_1)T_0(\mathbf{v}_1, \mathbf{v}_0) \quad (11.49)$$

$$\propto p(\mathbf{v}_n)p(\mathbf{v}_{n-1}|\mathbf{v}_n) \cdots p(\mathbf{v}_1|\mathbf{v}_0) \quad (11.50)$$

One can show that the importance weights $w = \frac{\varphi(\mathbf{v}_0, \dots, \mathbf{v}_n)}{g(\mathbf{v}_0, \dots, \mathbf{v}_n)}$ are given by Equation (11.45). Since marginals of the sampled sequences from this extended model are equivalent to samples from $p_0(\mathbf{x})$, we see that we are using the correct weights.

11.5.4.1 Estimating normalizing constants using AIS

An important application of AIS is to evaluate a ratio of partition functions. Notice that $Z_0 = \int f_0(\mathbf{x})d\mathbf{x} = \int \varphi(\mathbf{v})d\mathbf{v}$, and $Z_n = \int f_n(\mathbf{x})d\mathbf{x} = \int g(\mathbf{v})d\mathbf{v}$. Hence

$$\frac{Z_0}{Z_n} = \frac{\int \varphi(\mathbf{v})d\mathbf{v}}{\int g(\mathbf{v})d\mathbf{v}} = \frac{\int \frac{\varphi(\mathbf{v})}{g(\mathbf{v})}g(\mathbf{v})d\mathbf{v}}{\int g(\mathbf{v})d\mathbf{v}} = \mathbb{E}_g \left[\frac{\varphi(\mathbf{v})}{g(\mathbf{v})} \right] \approx \frac{1}{S} \sum_{s=1}^S w_s \quad (11.51)$$

where $w_s = \varphi(\mathbf{v}_s)/g(\mathbf{v}_s)$. If f_0 is a prior and f_n is the posterior, we can estimate $Z_n = p(\mathcal{D})$ using the above equation, provided the prior has a known normalization constant Z_0 . This is generally considered the method of choice for evaluating difficult partition functions. See e.g., [GM98] for more details.

11.6 Controlling Monte Carlo variance

As we mentioned in Section 11.2.2, the standard error in a Monte Carlo estimate is $O(1/\sqrt{S})$, where S is the number of (independent) samples. Consequently it may take many samples to reduce the variance to a sufficiently small value. In this section, we discuss some ways to reduce the variance of sampling methods. For more details, see e.g., [KTB11].

11.6.1 Common random numbers

When performing Monte Carlo optimization, we often want to compare $\mathbb{E}_{p(\mathbf{z})}[f(\boldsymbol{\theta}, \mathbf{z})]$ to $\mathbb{E}_{p(\mathbf{z})}[f(\boldsymbol{\theta}', \mathbf{z})]$ for different values of the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$. To reduce the variance of this comparison, we can use the same random samples \mathbf{z}_s for evaluating both functions. In this way, differences in the outcome can be ascribed to differences in the parameters $\boldsymbol{\theta}$, rather than to the noise terms. This is called the **common random numbers** trick, and is widely used in ML (see e.g., [GBJ18; NJ00]), since it can often convert a stochastic optimization problem into a deterministic one, enabling the use of more powerful optimization methods. For more details on CRN, see e.g., [https://en.wikipedia.org/wiki/Variance_reduction#Common_Random_Numbers_\(CRN\)](https://en.wikipedia.org/wiki/Variance_reduction#Common_Random_Numbers_(CRN)).

11.6.2 Rao-Blackwellization

In this section, we discuss a useful technique for reducing the variance of MC estimators known as **Rao-Blackwellization**. To explain the method, suppose we have two rv's, X and Y , and we want

to estimate $\bar{f} = \mathbb{E}[f(X, Y)]$. The naive approach is to use an MC approximation

$$\hat{f}_{MC} = \frac{1}{S} \sum_{s=1}^S f(X_s, Y_s) \quad (11.52)$$

where $(X_s, Y_s) \sim p(X, Y)$. This is an unbiased estimator of \bar{f} . However, it may have high variance.

Now suppose we can analytically marginalize out Y , provided we know X , i.e., we can tractably compute

$$f_X(X_s) = \int dY p(Y|X_s) f(X_s, Y) = \mathbb{E}[f(X, Y)|X = X_s] \quad (11.53)$$

Let us define the Rao-Blackwellized estimator

$$\hat{f}_{RB} = \frac{1}{S} \sum_{s=1}^S f_X(X_s) \quad (11.54)$$

where $X_s \sim p(X)$. This is an unbiased estimator, since $\mathbb{E}[\hat{f}_{RB}] = \mathbb{E}[\mathbb{E}[f(X, Y)|X]] = \bar{f}$. However, this estimate can have lower variance than the naive estimator. The intuitive reason is that we are now sampling in a reduced dimensional space. Formally we can see this by using the law of iterated variance to get

$$\mathbb{V}[\mathbb{E}[f(X, Y)|X]] = \mathbb{V}[f(X, Y)] - \mathbb{E}[\mathbb{V}[f(X, Y)]|X] \leq \mathbb{V}[f(X, Y)] \quad (11.55)$$

For some examples of this in practice, see Section 6.3.4.2, Section 13.4, and Section 12.3.8.

11.6.3 Control variates

Suppose we want to estimate $\mu = \mathbb{E}[f(X)]$ using an unbiased estimator $m(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S m(x_s)$, where $x_s \sim p(X)$ and $\mathbb{E}[m(X)] = \mu$. (We abuse notation slightly and use m to refer to a function of a single random variable as well as a set of samples.) Now consider the alternative estimator

$$m^*(\mathcal{X}) = m(\mathcal{X}) + c(b(\mathcal{X}) - \mathbb{E}[b(\mathcal{X})]) \quad (11.56)$$

This is called a **control variate**, and b is called a **baseline**. (Once again we abuse notation and use $b(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S b(x_s)$ and $m^*(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S m^*(x_s)$.)

It is easy to see that $m^*(\mathcal{X})$ is an unbiased estimator, since $\mathbb{E}[m^*(X)] = \mathbb{E}[m(X)] = \mu$. However, it can have lower variance, provided b is correlated with m . To see this, note that

$$\mathbb{V}[m^*(X)] = \mathbb{V}[m(X)] + c^2 \mathbb{V}[b(X)] + 2c \text{Cov}[m(X), b(X)] \quad (11.57)$$

By taking the derivative of $\mathbb{V}[m^*(X)]$ wrt c and setting to 0, we find that the optimal value is

$$c^* = -\frac{\text{Cov}[m(X), b(X)]}{\mathbb{V}[b(X)]} \quad (11.58)$$

The corresponding variance of the new estimator is now

$$\mathbb{V}[m^*(X)] = \mathbb{V}[m(X)] - \frac{\text{Cov}[m(X), b(X)]^2}{\mathbb{V}[b(X)]} = (1 - \rho_{m,b}^2)\mathbb{V}[m(X)] \leq \mathbb{V}[m(X)] \quad (11.59)$$

where $\rho_{m,b}^2$ is the correlation of the basic estimator and the baseline function. If we can ensure this correlation is high, we can reduce the variance. Intuitively, the CV estimator is exploiting information about the errors in the estimate of a known quantity, namely $\mathbb{E}[b(X)]$, to reduce the errors in estimating the unknown quantity, namely μ .

We give a simple worked example in Section 11.6.3.1. See Section 10.2.3 for an example of this technique applied to blackbox variational inference.

11.6.3.1 Example

We now give a simple worked example of control variates.⁴ Consider estimating $\mu = \mathbb{E}[f(X)]$ where $f(X) = 1/(1+X)$ and $X \sim \text{Unif}(0, 1)$. The exact value is

$$\mu = \int_0^1 \frac{1}{1+x} dx = \ln 2 \approx 0.693 \quad (11.60)$$

The naive MC estimate, using S samples, is $m(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S f(x_s)$. Using $S = 1500$, we find $\mathbb{E}[m(\mathcal{X})] = 0.6935$ with standard error $\text{se} = 0.0037$.

Now let us use $b(X) = 1 + X$ as a baseline, so $b(\mathcal{X}) = (1/S) \sum_s (1 + x_s)$. This has expectation $\mathbb{E}[b(X)] = \int_0^1 (1+x) dx = \frac{3}{2}$. The control variate estimator is given by

$$m^*(\mathcal{X}) = \frac{1}{S} \sum_{s=1}^S f(x_s) + c \left(\frac{1}{S} \sum_{s=1}^S b(x_s) - \frac{3}{2} \right) \quad (11.61)$$

The optimal value can be estimated from the samples of $m(x_s)$ and $b(x_s)$, and plugging into Equation (11.58) to get $c^* \approx 0.4773$. Using $S = 1500$, we find $\mathbb{E}[m^*(\mathcal{X})] = 0.6941$ and $\text{se} = 0.0007$.

See also Section 11.6.4.1, where we analyze this example using antithetic sampling.

11.6.4 Antithetic sampling

In this section, we discuss **antithetic sampling**, which is a simple way to reduce variance.⁵ Suppose we want to estimate $\theta = \mathbb{E}[Y]$. Let Y_1 and Y_2 be two samples. An unbiased estimate of θ is given by $\hat{\theta} = (Y_1 + Y_2)/2$. The variance of this estimate is

$$\mathbb{V}[\hat{\theta}] = \frac{\mathbb{V}[Y_1] + \mathbb{V}[Y_2] + 2\text{Cov}[Y_1, Y_2]}{4} \quad (11.62)$$

so the variance is reduced if $\text{Cov}[Y_1, Y_2] < 0$. So whenever we sample Y_1 , we should set Y_2 to be its “opposite”, but with the same mean.

For example, suppose $Y \sim \text{Unif}(0, 1)$. If we let y_1, \dots, y_n be iid samples from $\text{Unif}(0, 1)$, then we can define $y'_i = 1 - y_i$. The distribution of y'_i is still $\text{Unif}(0, 1)$, but $\text{Cov}[y_i, y'_i] < 1$.

4. The example is from https://en.wikipedia.org/wiki/Control_variates, with modified notation. See `control_variates.ipynb` for some code.

5. Our presentation is based on https://en.wikipedia.org/wiki/Antithetic_variates. See `antithetic_sampling.ipynb` for the code.

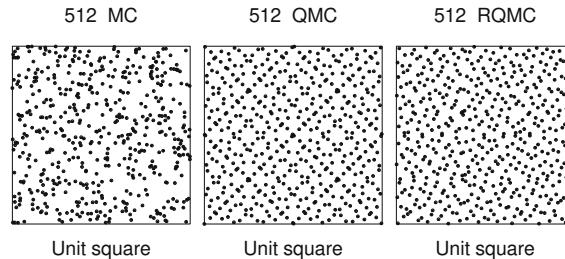


Figure 11.7: Illustration of Monte Carlo (MC), Quasi-MC (QMC) from a Sobol sequence, and randomized QMC using a scrambling method. Adapted from Figure 1 of [OR20]. Used with kind permission of Art Owen.

11.6.4.1 Example

To see why this can be useful, consider the example from Section 11.6.3.1. Let $\hat{\mu}_{\text{mc}}$ be the classic MC estimate using $2N$ samples from $\text{Unif}(0, 1)$, and let $\hat{\mu}_{\text{anti}}$ be the MC estimate using the above antithetic sampling scheme applied to N base samples from $\text{Unif}(0, 1)$. The exact value is $\mu = \ln 2 \approx 0.6935$. For the classical method, with $N = 750$, we find $\mathbb{E}[\hat{\mu}_{\text{mc}}] = 0.69365$ with a standard error of 0.0037. For the antithetic method, we find $\mathbb{E}[\hat{\mu}_{\text{anti}}] = 0.6939$ with a standard error of 0.0007, which matches the control variate method of Section 11.6.3.1.

11.6.5 Quasi-Monte Carlo (QMC)

Quasi-Monte Carlo (see e.g., [Lem09; Owe13]) is an approach to numerical integration that replaces random samples with **low discrepancy sequences**, such as the **Halton sequence** (see e.g., [Owe17]) or **Sobol sequence**. Intuitively, these are **space filling** sequences of points, constructed to reduce the unwanted gaps and clusters that would arise among randomly chosen inputs. See Figure 11.7 for an example.⁶

More precisely, consider the problem of evaluating the following D -dimensional integral:

$$\bar{f} = \int_{[0,1]^D} f(\mathbf{x}) d\mathbf{x} \approx \hat{f}_N = \frac{1}{N} \sum_{n=1}^N f(\mathbf{x}_n) \quad (11.63)$$

Let $\epsilon_N = |\bar{f} - \hat{f}_N|$ be the error. In standard Monte Carlo, if we draw N independent samples, then we have $\epsilon_N \sim O\left(\frac{1}{\sqrt{N}}\right)$. In QMC, it can be shown that $\epsilon_N \sim O\left(\frac{(\log N)^D}{N}\right)$. For $N > 2^D$, the latter is smaller than the former.

One disadvantage of QMC is that it just provides a point estimate of \bar{f} , and does not give an uncertainty estimate. By contrast, in regular MC, we can estimate the MC standard error, discussed in Section 11.2.2. **Randomized QMC** (see e.g., [L'E18]) provides a solution to this problem. The basic idea is to repeat the QMC method R times, by perturbing the sequence of N points by a

6. More details on QMC can be found at http://roth.cs.kuleuven.be/wiki/Main_Page. For connections to Bayesian quadrature, see e.g., [DKS13; HKO22].

random amount. In particular, define

$$\mathbf{y}_{i,r} = \mathbf{x}_i + \mathbf{u}_r \pmod{1} \quad (11.64)$$

where $\mathbf{x}_1, \dots, \mathbf{x}_N$ is a low-discrepancy sequence, and $\mathbf{u}_r \sim \text{Unif}(0, 1)^D$ is a random perturbation. The set $\{\mathbf{y}_j\}$ is low discrepancy, and satisfies that each $\mathbf{y}_j \sim \text{Unif}(0, 1)^D$, for $j = 1 : N \times R$. This has much lower variance than standard MC. (Typically we take R to be a power of 2.) Recently, [OR20] proved a strong law of large numbers for RQMC.

QMC and RQMC can be used inside of MCMC inference (see e.g., [OT05]) and variational inference (see e.g., [BWM18]). It is also commonly used to select the initial set of query points for Bayesian optimization (Section 6.6).

Another technique that can be used is **orthogonal Monte Carlo**, where the samples are conditioned to be pairwise orthogonal, but with the marginal distributions matching the original ones (see e.g., [Lin+20]).

12 Markov chain Monte Carlo

12.1 Introduction

In Chapter 11, we considered non-iterative Monte Carlo methods, including rejection sampling and importance sampling, which generate independent samples from some target distribution. The trouble with these methods is that they often do not work well in high dimensional spaces. In this chapter, we discuss a popular method for sampling from high-dimensional distributions known as **Markov chain Monte Carlo** or **MCMC**. In a survey by *SIAM News*¹, MCMC was placed in the top 10 most important algorithms of the 20th century.

The basic idea behind MCMC is to construct a Markov chain (Section 2.6) on the state space \mathcal{X} whose stationary distribution is the target density $p^*(\mathbf{x})$ of interest. (In a Bayesian context, this is usually a posterior, $p^*(\mathbf{x}) \propto p(\mathbf{x}|\mathcal{D})$, but MCMC can be applied to generate samples from any kind of distribution.) That is, we perform a random walk on the state space, in such a way that the fraction of time we spend in each state \mathbf{x} is proportional to $p^*(\mathbf{x})$. By drawing (correlated) samples $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$, from the chain, we can perform Monte Carlo integration wrt p^* .

Note that the initial samples from the chain do not come from the stationary distribution, and should be discarded; the amount of time it takes to reach stationarity is called the **mixing time** or **burn-in time**; reducing this is one of the most important factors in making the algorithm fast, as we will see.

The MCMC algorithm has an interesting history. It was discovered by physicists working on the atomic bomb at Los Alamos during World War II, and was first published in the open literature in [Met+53] in a chemistry journal. An extension was published in the statistics literature in [Has70], but was largely unnoticed. A special case (Gibbs sampling, Section 12.3) was independently invented in [GG84] in the context of Ising models (Section 4.3.2.1). But it was not until [GS90] that the algorithm became well-known to the wider statistical community. Since then it has become wildly popular in Bayesian statistics, and is becoming increasingly popular in machine learning.

In the rest of this chapter, we give a brief introduction to MCMC methods. For more details on the theory, see e.g., [GRS96; BZ20]. For more details on the implementation side, see e.g., [Lao+20]. And for an interactive visualization of many of these algorithms in 2d, see <http://chi-feng.github.io/mcmc-demo/app.html>.

1. Source: <http://www.siam.org/pdf/news/637.pdf>.

12.2 Metropolis-Hastings algorithm

In this section, we describe the simplest kinds of MCMC algorithm known as the **Metropolis-Hastings** or **MH** algorithm.

12.2.1 Basic idea

The basic idea in MH is that at each step, we propose to move from the current state \mathbf{x} to a new state \mathbf{x}' with probability $q(\mathbf{x}'|\mathbf{x})$, where q is called the **proposal distribution** (also called the **kernel**). The user is free to use any kind of proposal they want, subject to some conditions which we explain below. This makes MH quite a flexible method.

Having proposed a move to \mathbf{x}' , we then decide whether to **accept** this proposal, or to reject it, according to some formula, which ensures that the long-term fraction of time spent in each state is proportional to $p^*(\mathbf{x})$. If the proposal is accepted, the new state is \mathbf{x}' , otherwise the new state is the same as the current state, \mathbf{x} (i.e., we repeat the sample).

If the proposal is symmetric, so $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}|\mathbf{x}')$, the acceptance probability is given by the following formula:

$$A = \min \left(1, \frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})} \right) \quad (12.1)$$

We see that if \mathbf{x}' is more probable than \mathbf{x} , we definitely move there (since $\frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})} > 1$), but if \mathbf{x}' is less probable, we may still move there anyway, depending on the relative probabilities. So instead of greedily moving to only more probable states, we occasionally allow “downhill” moves to less probable states. In Section 12.2.2, we prove that this procedure ensures that the fraction of time we spend in each state \mathbf{x} is equal to $p^*(\mathbf{x})$.

If the proposal is asymmetric, so $q(\mathbf{x}'|\mathbf{x}) \neq q(\mathbf{x}|\mathbf{x}')$, we need the **Hastings correction**, given by the following:

$$A = \min(1, \alpha) \quad (12.2)$$

$$\alpha = \frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{p^*(\mathbf{x}')/q(\mathbf{x}'|\mathbf{x})}{p^*(\mathbf{x})/q(\mathbf{x}|\mathbf{x}')} \quad (12.3)$$

This correction is needed to compensate for the fact that the proposal distribution itself (rather than just the target distribution) might favor certain states.

An important reason why MH is a useful algorithm is that, when evaluating α , we only need to know the target density up to a normalization constant. In particular, suppose $p^*(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x})$, where $\tilde{p}(\mathbf{x})$ is an unnormalized distribution and Z is the normalization constant. Then

$$\alpha = \frac{(\tilde{p}(\mathbf{x}')/Z) q(\mathbf{x}|\mathbf{x}')}{(\tilde{p}(\mathbf{x})/Z) q(\mathbf{x}'|\mathbf{x})} \quad (12.4)$$

so the Z 's cancel. Hence we can sample from p^* even if Z is unknown.

A proposal distribution q is valid or admissible if it “covers” the support of the target. Formally, we can write this as

$$\text{supp}(p^*) \subseteq \cup_x \text{supp}(q(\cdot|x)) \quad (12.5)$$

With this, we can state the overall algorithm as in Algorithm 12.1.

Algorithm 12.1: Metropolis-Hastings algorithm

```

1 Initialize  $x^0$ 
2 for  $s = 0, 1, 2, \dots$  do
3   Define  $x = x^s$ 
4   Sample  $x' \sim q(x'|x)$ 
5   Compute acceptance probability
      
$$\alpha = \frac{\tilde{p}(x')q(x|x')}{\tilde{p}(x)q(x'|x)}$$

6   Compute  $A = \min(1, \alpha)$ 
7   Sample  $u \sim U(0, 1)$ 
8   Set new sample to
      
$$x^{s+1} = \begin{cases} x' & \text{if } u \leq A \text{ (accept)} \\ x^s & \text{if } u > A \text{ (reject)} \end{cases}$$


```

12.2.2 Why MH works

To prove that the MH procedure generates samples from p^* , we need a bit of Markov chain theory, as discussed in Section 2.6.4.

The MH algorithm defines a Markov chain with the following transition matrix:

$$p(\mathbf{x}'|\mathbf{x}) = \begin{cases} q(\mathbf{x}'|\mathbf{x})A(\mathbf{x}'|\mathbf{x}) & \text{if } \mathbf{x}' \neq \mathbf{x} \\ q(\mathbf{x}|\mathbf{x}) + \sum_{\mathbf{x}' \neq \mathbf{x}} q(\mathbf{x}'|\mathbf{x})(1 - A(\mathbf{x}'|\mathbf{x})) & \text{otherwise} \end{cases} \quad (12.6)$$

This follows from a case analysis: if you move to \mathbf{x}' from \mathbf{x} , you must have proposed it (with probability $q(\mathbf{x}'|\mathbf{x})$) and it must have been accepted (with probability $A(\mathbf{x}'|\mathbf{x})$); otherwise you stay in state \mathbf{x} , either because that is what you proposed (with probability $q(\mathbf{x}|\mathbf{x})$), or because you proposed something else (with probability $q(\mathbf{x}'|\mathbf{x})$) but it was rejected (with probability $1 - A(\mathbf{x}'|\mathbf{x})$).

Let us analyze this Markov chain. Recall that a chain satisfies **detailed balance** if

$$p(\mathbf{x}'|\mathbf{x})p^*(\mathbf{x}) = p(\mathbf{x}|\mathbf{x}')p^*(\mathbf{x}') \quad (12.7)$$

This means in the in-flow to state \mathbf{x}' from \mathbf{x} is equal to the out-flow from state \mathbf{x}' back to \mathbf{x} , and vice versa. We also showed that if a chain satisfies detailed balance, then p^* is its stationary distribution. Our goal is to show that the MH algorithm defines a transition function that satisfies detailed balance and hence that p^* is its stationary distribution. (If Equation (12.7) holds, we say that p^* is an **invariant** distribution wrt the Markov transition kernel q .)

Theorem 12.2.1. *If the transition matrix defined by the MH algorithm (given by Equation (12.6)) is ergodic and irreducible, then p^* is its unique limiting distribution.*

Proof. Consider two states \mathbf{x} and \mathbf{x}' . Either

$$p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) < p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}') \quad (12.8)$$

or

$$p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) \geq p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}') \quad (12.9)$$

Without loss of generality, assume that $p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x}) > p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')$. Hence

$$\alpha(\mathbf{x}'|\mathbf{x}) = \frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} < 1 \quad (12.10)$$

Hence we have $A(\mathbf{x}'|\mathbf{x}) = \alpha(\mathbf{x}'|\mathbf{x})$ and $A(\mathbf{x}|\mathbf{x}') = 1$.

Now to move from \mathbf{x} to \mathbf{x}' we must first propose \mathbf{x}' and then accept it. Hence

$$p(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}'|\mathbf{x})A(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}'|\mathbf{x})\frac{p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{p^*(\mathbf{x})q(\mathbf{x}'|\mathbf{x})} = \frac{p^*(\mathbf{x}')}{p^*(\mathbf{x})}q(\mathbf{x}|\mathbf{x}') \quad (12.11)$$

Hence

$$p^*(\mathbf{x})p(\mathbf{x}'|\mathbf{x}) = p^*(\mathbf{x}')q(\mathbf{x}|\mathbf{x}') \quad (12.12)$$

The backwards probability is

$$p(\mathbf{x}|\mathbf{x}') = q(\mathbf{x}|\mathbf{x}')A(\mathbf{x}|\mathbf{x}') = q(\mathbf{x}|\mathbf{x}') \quad (12.13)$$

since $A(\mathbf{x}|\mathbf{x}') = 1$. Inserting this into Equation (12.12) we get

$$p^*(\mathbf{x})p(\mathbf{x}'|\mathbf{x}) = p^*(\mathbf{x}')p(\mathbf{x}|\mathbf{x}') \quad (12.14)$$

so detailed balance holds wrt p^* . Hence, from Theorem 2.6.3, p^* is a stationary distribution. Furthermore, from Theorem 2.6.2, this distribution is unique, since the chain is ergodic and irreducible. \square

12.2.3 Proposal distributions

In this section, we discuss some common proposal distributions. Note, however, that good proposal design is often intimately dependent on the form of the target distribution (most often the posterior).

12.2.3.1 Independence sampler

If we use a proposal of the form $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}')$, where the new state is independent of the old state, we get a method known as the **independence sampler**, which is similar to importance sampling (Section 11.5). The function $q(\mathbf{x}')$ can be any suitable distribution, such as a Gaussian. This has non-zero probability density on the entire state space, and hence is a valid proposal for any unconstrained continuous state space.

12.2.3.2 Random walk Metropolis (RWM) algorithm

The **random walk Metropolis** algorithm corresponds to MH with the following proposal distribution:

$$q(\mathbf{x}'|\mathbf{x}) = \mathcal{N}(\mathbf{x}'|\mathbf{x}, \tau^2 \mathbf{I}) \quad (12.15)$$

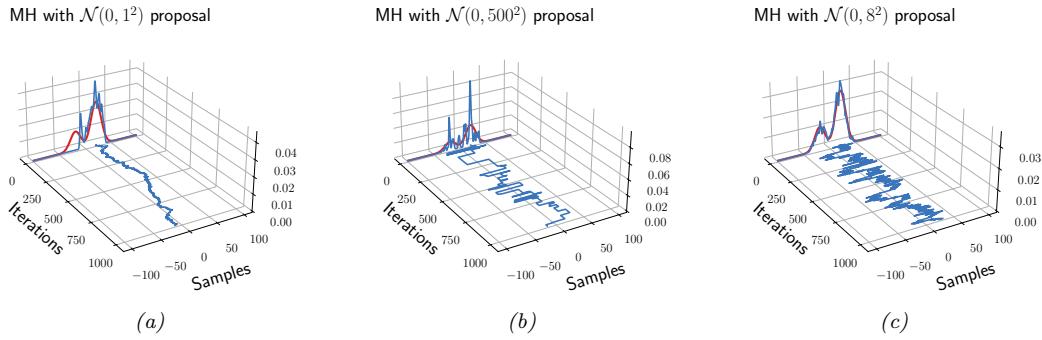


Figure 12.1: An example of the Metropolis-Hastings algorithm for sampling from a mixture of two 1D Gaussians ($\mu = (-20, 20)$, $\pi = (0.3, 0.7)$, $\Sigma = (100, 100)$), using a Gaussian proposal with standard deviation of $\tau \in \{1, 8, 500\}$. (a) When $\tau = 1$, the chain gets trapped near the starting state and fails to sample from the mode at $\mu = -20$. (b) When $\tau = 500$, the chain is very “sticky”, so its effective sample size is low (as reflected by the rough histogram approximation at the end). (c) Using a variance of $\tau = 8$ is just right and leads to a good approximation of the true distribution (shown in red). Compare to Figure 12.4. Generated by `mcmc_gmm_demo.ipynb`.

Here τ is a scale factor chosen to facilitate rapid mixing. [RR01b] prove that, if the posterior is Gaussian, the asymptotically optimal value is to use $\tau^2 = 2.38^2/D$, where D is the dimensionality of \mathbf{x} ; this results in an acceptance rate of 0.234, which (in this case) is the optimal tradeoff between exploring widely enough to cover the distribution without being rejected too often. (See [Béd08] for a more recent account of optimal acceptance rates for random walk Metropolis methods.)

Figure 12.1 shows an example where we use RWM to sample from a mixture of two 1D Gaussians. This is a somewhat tricky target distribution, since it consists of two somewhat separated modes. It is very important to set the variance of the proposal τ^2 correctly: if the variance is too low, the chain will only explore one of the modes, as shown in Figure 12.1(a), but if the variance is too large, most of the moves will be rejected, and the chain will be very **sticky**, i.e., it will stay in the same state for a long time. This is evident from the long stretches of repeated values in Figure 12.1(b). If we set the proposal’s variance just right, we get the trace in Figure 12.1(c), where the samples clearly explore the support of the target distribution.

12.2.3.3 Composing proposals

If there are several proposals that might be useful, one can combine them using a **mixture proposal**, which is a convex combination of base proposals:

$$q(\mathbf{x}'|\mathbf{x}) = \sum_{k=1}^K w_k q_k(\mathbf{x}'|\mathbf{x}) \quad (12.16)$$

where w_k are the mixing weights that sum to one. As long as each q_k is an individually valid proposal, and each $w_k > 0$, then the overall mixture proposal will also be valid. In particular, if each proposal is reversible, so it satisfies detailed balance (Section 2.6.4.4), then so does the mixture.

It is also possible to compose individual proposals by chaining them together to get

$$q(\mathbf{x}'|\mathbf{x}) = \sum_{\mathbf{x}_1} \cdots \sum_{\mathbf{x}_{K-1}} q_1(\mathbf{x}_1|\mathbf{x}) q_2(\mathbf{x}_2|\mathbf{x}_1) \cdots q_K(\mathbf{x}|\mathbf{x}_{K-1}) \quad (12.17)$$

A common example is where each base proposal only updates a subset of the variables (see e.g., Section 12.3).

12.2.3.4 Data-driven MCMC

In the case where the target distribution is a posterior, $p^*(\mathbf{x}) = p(\mathbf{x}|\mathcal{D})$, it is helpful to condition the proposal not just on the previous hidden state, but also the visible data, i.e., to use $q(\mathbf{x}'|\mathbf{x}, \mathcal{D})$. This is called **data-driven MCMC** (see e.g., [TZ02; Jih+12]).

One way to create such a proposal is to train a recognition network to propose states using $q(\mathbf{x}'|\mathbf{x}, \mathcal{D}) = f(\mathbf{x})$. If the state space is high-dimensional, it might be hard to predict all the hidden components, so we can alternatively train individual “experts” to predict specific pieces of the hidden state. For example, in the context of estimating the 3d pose of a person from an image, we might combine a face detector with a limb detector. We can then use a mixture proposal of the form

$$q(\mathbf{x}'|\mathbf{x}, \mathcal{D}) = \pi_0 q_0(\mathbf{x}'|\mathbf{x}) + \sum_k \pi_k q_k(x'_k|f_k(\mathcal{D})) \quad (12.18)$$

where q_0 is a standard data-independent proposal (e.g., random walk), and q_k updates the k 'th component of the state space.

The overall procedure is a form of **generate and test**: the discriminative proposals $q(\mathbf{x}'|\mathbf{x}, \mathcal{D})$ generate new hypotheses, which are then “tested” by computing the posterior ratio $\frac{p(\mathbf{x}'|\mathcal{D})}{p(\mathbf{x}|\mathcal{D})}$, to see if the new hypothesis is better or worse. (See also Section 13.3, where we discuss learning proposal distributions for particle filters.)

12.2.3.5 Adaptive MCMC

One can change the parameters of the proposal as the algorithm is running to increase efficiency. This is called **adaptive MCMC**. This allows one to start with a broad covariance (say), allowing large moves through the space until a mode is found, followed by a narrowing of the covariance to ensure careful exploration of the region around the mode.

However, one must be careful not to violate the Markov property; thus the parameters of the proposal should not depend on the entire history of the chain. It turns out that a sufficient condition to ensure this is that the adaption is “faded out” gradually over time. See e.g., [AT08] for details.

12.2.4 Initialization

It is necessary to start MCMC in an initial state that has non-zero probability. A natural approach is to first use an optimizer to find a local mode. However, at such points the gradients of the log joint are zero, which can cause problems for some gradient-based MCMC methods, such as HMC (Section 12.5), so it can be better to start “close” to a MAP estimate (see e.g., [HFM17, Sec 7.]).

12.3 Gibbs sampling

The major problems with MH are the need to choose the proposal distribution, and the fact that the acceptance rate may be low. In this section, we describe an MH method that exploits conditional independence properties of a graphical model to automatically create a good proposal, with acceptance probability 1. This method is known as **Gibbs sampling**.² (In physics, this method is known as **Glauber dynamics** or the **heat bath** method.) This is the MCMC analog of coordinate descent.³

12.3.1 Basic idea

The idea behind Gibbs sampling is to sample each variable in turn, conditioned on the values of all the other variables in the distribution. For example, if we have $D = 3$ variables, we use

- $x_1^{s+1} \sim p(x_1|x_2^s, x_3^s)$
- $x_2^{s+1} \sim p(x_2|x_1^{s+1}, x_3^s)$
- $x_3^{s+1} \sim p(x_3|x_1^{s+1}, x_2^{s+1})$

This readily generalizes to D variables. (Note that if x_i is a known variable, we do not sample it, but it may be used as input to the another conditional distribution.)

The expression $p(x_i|\mathbf{x}_{-i})$ is called the **full conditional** for variable i . In general, x_i may only depend on some of the other variables. If we represent $p(\mathbf{x})$ as a graphical model, we can infer the dependencies by looking at i 's Markov blanket, which are its neighbors in the graph (see Section 4.2.4.3), so we can write

$$x_i^{s+1} \sim p(x_i|\mathbf{x}_{-i}) = p(x_i|\mathbf{x}_{\text{mb}(i)}) \quad (12.19)$$

(Compare to the equation for mean field variational inference in Equation (10.87).)

We can sample some of the nodes in parallel, without affecting correctness. In particular, suppose we can create a **coloring** of the (moralized) undirected graph, such that no two neighboring nodes have the same color. (In general, computing an optimal coloring is NP-complete, but we can use efficient heuristics such as those in [Kub04].) Then we can sample all the nodes of the same color in parallel, and cycle through the colors sequentially [Gon+11].

12.3.2 Gibbs sampling is a special case of MH

It turns out that Gibbs sampling is a special case of MH where we use a sequence of proposals of the form

$$q_i(\mathbf{x}'|\mathbf{x}) = p(x'_i|\mathbf{x}_{-i})\mathbb{I}(x'_{-i} = \mathbf{x}_{-i}) \quad (12.20)$$

That is, we move to a new state where x_i is sampled from its full conditional, but \mathbf{x}_{-i} is left unchanged.

2. Josiah Willard Gibbs, 1839–1903, was an American physicist.

3. Several software libraries exist for applying Gibbs sampling to general graphical models, including [Nimble](#), which is a C++ library with an R wrapper, and which replaces older programs such as BUGS and JAGS.

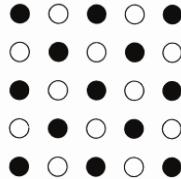


Figure 12.2: Illustration of checkerboard pattern for a 2d MRF. This allows for parallel updates.

We now prove that the acceptance rate of each such proposal is 100%, so the overall algorithm also has an acceptance rate of 100%. We have

$$\alpha = \frac{p(\mathbf{x}')q_i(\mathbf{x}|\mathbf{x}')}{p(\mathbf{x})q_i(\mathbf{x}'|\mathbf{x})} = \frac{p(x'_i|x'_{-i})p(x'_{-i})p(x_i|\mathbf{x}'_{-i})}{p(x_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x'_i|\mathbf{x}_{-i})} \quad (12.21)$$

$$= \frac{p(x'_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x_i|\mathbf{x}_{-i})}{p(x_i|\mathbf{x}_{-i})p(\mathbf{x}_{-i})p(x'_i|\mathbf{x}_{-i})} = 1 \quad (12.22)$$

where we exploited the fact that $\mathbf{x}'_{-i} = \mathbf{x}_{-i}$.

The fact that the acceptance rate is 100% does not necessarily mean that Gibbs will converge rapidly, since it only updates one coordinate at a time (see Section 12.3.7). However, if we can group together correlated variables, then we can sample them as a group, which can significantly help mixing.

12.3.3 Example: Gibbs sampling for Ising models

In Section 4.3.2.1, we discuss Ising models and Potts models, which are pairwise MRFs with a 2d grid structure. The joint distribution has the form

$$p(\mathbf{x}) = \frac{1}{Z} \prod_{i \sim j} \psi_{ij}(x_i, x_j | \boldsymbol{\theta}) \quad (12.23)$$

where $i \sim j$ means i and j are neighbors in the graph.

To apply Gibbs sampling to such a model, we just need to iteratively sample from each full conditional:

$$p(x_i | \mathbf{x}_{-i}) \propto \prod_{j \in \text{nbr}(i)} \psi_{ij}(x_i, x_j) \quad (12.24)$$

Note that although Gibbs sampling is a sequential algorithm, we can sometimes exploit conditional independence properties to perform parallel updates [RS97a]. In the case of a 2d grid, we can color code nodes using a checkerboard pattern shown in Figure 12.2. This has the property that the black nodes are conditionally independent of each other given the white nodes, and vice versa. Hence we can sample all the black nodes in parallel (as a single group), and then sample all the white nodes, etc.

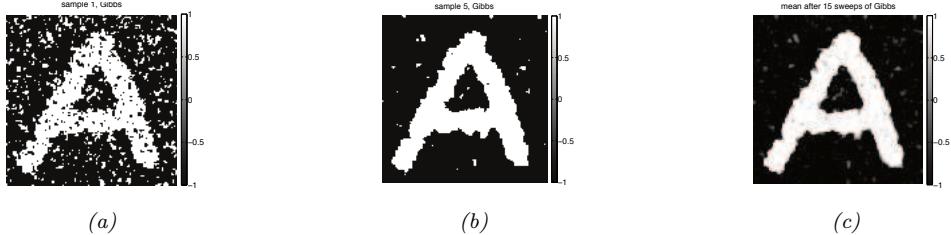


Figure 12.3: Example of image denoising using Gibbs sampling. We use an Ising prior with $J = 1$ and a Gaussian noise model with $\sigma = 2$. (a) Sample from the posterior after one sweep over the image. (b) Sample after 5 sweeps. (c) Posterior mean, computed by averaging over 15 sweeps. Compare to Figure 10.9 which shows the results of mean field inference. Generated by [ising_image_denoise_demo.ipynb](#).

To perform the sampling, we need to compute the full conditional in Equation (12.24). In the case of an Ising model with edge potentials $\psi(x_i, x_j) = \exp(Jx_i x_j)$, where $x_i \in \{-1, +1\}$, the full conditional becomes

$$p(x_i = +1 | \mathbf{x}_{-i}) = \frac{\prod_{j \in \text{nbr}(i)} \psi_{ij}(x_i = +1, x_j)}{\prod_{j \in \text{nbr}(i)} \psi(x_i = +1, x_j) + \prod_{j \in \text{nbr}(i)} \psi(x_i = -1, x_j)} \quad (12.25)$$

$$= \frac{\exp[J \sum_{j \in \text{nbr}(i)} x_j]}{\exp[J \sum_{j \in \text{nbr}(i)} x_j] + \exp[-J \sum_{j \in \text{nbr}(i)} x_j]} \quad (12.26)$$

$$= \frac{\exp[J\eta_i]}{\exp[J\eta_i] + \exp[-J\eta_i]} = \sigma(2J\eta_i) \quad (12.27)$$

where J is the coupling strength, $\eta_i \triangleq \sum_{j \in \text{nbr}(i)} x_j$, and $\sigma(u) = 1/(1 + e^{-u})$ is the sigmoid function. (If we use $x_i \in \{0, 1\}$, this becomes $p(x_i = +1 | \mathbf{x}_{-i}) = \sigma(J\eta_i)$.) It is easy to see that $\eta_i = x_i(a_i - d_i)$, where a_i is the number of neighbors that agree with (have the same sign as) node i , and d_i is the number of neighbors who disagree. If this number is equal, the “forces” on x_i cancel out, so the full conditional is uniform. Some samples from this model are shown in Figure 4.17.

One application of Ising models is as a prior for binary image denoising problems. In particular, suppose \mathbf{y} is a noisy version of \mathbf{x} , and we wish to compute the posterior $p(\mathbf{x} | \mathbf{y}) \propto p(\mathbf{x})p(\mathbf{y} | \mathbf{x})$, where $p(\mathbf{x})$ is an Ising prior, and $p(\mathbf{y} | \mathbf{x}) = \prod_i p(y_i | x_i)$ is a per-site likelihood term. Suppose this is a Gaussian. Let $\psi_i(x_i) = \mathcal{N}(y_i | x_i, \sigma^2)$ be the corresponding “local evidence” term. The full conditional becomes

$$p(x_i = +1 | \mathbf{x}_{-i}, \mathbf{y}) = \frac{\exp[J\eta_i]\psi_i(+1)}{\exp[J\eta_i]\psi_i(+1) + \exp[-J\eta_i]\psi_i(-1)} \quad (12.28)$$

$$= \sigma\left(2J\eta_i - \log \frac{\psi_i(+1)}{\psi_i(-1)}\right) \quad (12.29)$$

Now the probability of x_i entering each state is determined both by compatibility with its neighbors (the Ising prior) and compatibility with the data (the local likelihood term).

See Figure 12.3 for an example of this algorithm applied to a simple image denoising problem. The results are similar to the mean field results in Figure 10.9.

12.3.4 Example: Gibbs sampling for Potts models

We can extend Section 12.3.3 to the Potts models as follows. Recall that the model has the following form:

$$p(\mathbf{x}) = \frac{1}{Z} \exp(-\mathcal{E}(\mathbf{x})) \quad (12.30)$$

$$\mathcal{E}(\mathbf{x}) = -J \sum_{i \sim j} \mathbb{I}(x_i = x_j) \quad (12.31)$$

For a node i with neighbors $\text{nbr}(i)$, the full conditional is thus given by

$$p(x_i = k | \mathbf{x}_{-i}) = \frac{\exp(J \sum_{n \in \text{nbr}(i)} \mathbb{I}(x_n = k))}{\sum_{k'} \exp(J \sum_{n \in \text{nbr}(i)} \mathbb{I}(x_n = k'))} \quad (12.32)$$

So if $J > 0$, a node i is more likely to enter a state k if most of its neighbors are already in state k , corresponding to an attractive MRF. If $J < 0$, a node i is more likely to enter a different state from its neighbors, corresponding to a repulsive MRF. See Figure 4.18 for some samples from this model created using this method.

12.3.5 Example: Gibbs sampling for GMMs

In this section, we consider sampling from a Bayesian Gaussian mixture model of the form

$$p(z = k, \mathbf{x} | \boldsymbol{\theta}) = \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.33)$$

$$p(\boldsymbol{\theta}) = \text{Dir}(\boldsymbol{\pi} | \boldsymbol{\alpha}) \prod_{k=1}^K \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_0, \mathbf{V}_0) \text{IW}(\boldsymbol{\Sigma}_k, \mathbf{S}_0, \nu_0) \quad (12.34)$$

12.3.5.1 Known parameters

Suppose, initially, that the parameters $\boldsymbol{\theta}$ are known. We can easily draw independent samples from $p(\mathbf{x} | \boldsymbol{\theta})$ by using ancestral sampling: first sample z and then \mathbf{x} . However, for illustrative purposes, we will use Gibbs sampling to draw correlated samples. The full conditional for $p(\mathbf{x} | z = k, \boldsymbol{\theta})$ is just $\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, and the full conditional for $p(z = k | \mathbf{x})$ is given by Bayes' rule:

$$p(z = k | \mathbf{x}, \boldsymbol{\theta}) = \frac{\pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} \quad (12.35)$$

An example of this procedure, applied to a mixture of two 1d Gaussians with means at -20 and $+20$, is shown in Figure 12.4. We see that the samples are auto correlated, meaning that if we are in state 1, we will likely stay in that state for a while, and generate values near μ_1 ; then we will stochastically jump to state 2, and stay near there for a while, etc. (See Section 12.6.3 for a way to measure this.) By contrast, independent samples from the joint would not be correlated at all.

In Section 12.3.5.2, we modify this example to sample the parameters of the GMM from their posterior, $p(\boldsymbol{\theta} | \mathcal{D})$, instead of sampling from $p(\mathcal{D} | \boldsymbol{\theta})$.

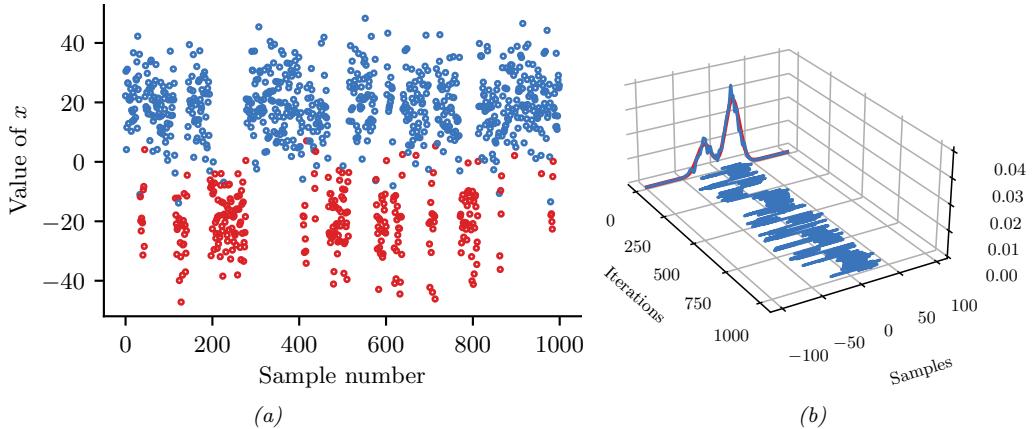


Figure 12.4: (a) Some samples from a mixture of two 1d Gaussians generated using Gibbs sampling. Color denotes the value of z , vertical location denotes the value of x . Horizontal axis represents time (sample number). (b) Traceplot of x over time, and the resulting empirical distribution is shown in blue. The true distribution is shown in red. Compare to Figure 12.1. Generated by `mcmc_gmm_demo.ipynb`.

12.3.5.2 Unknown parameters

Now suppose the parameters are unknown, so we want to fit the model to data. If we use a conditionally conjugate factored prior, then the full joint distribution is given by

$$p(\mathbf{x}, \mathbf{z}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu}, \boldsymbol{\Sigma})p(\mathbf{z}|\boldsymbol{\pi})p(\boldsymbol{\pi}) \prod_{k=1}^K p(\boldsymbol{\mu}_k)p(\boldsymbol{\Sigma}_k) \quad (12.36)$$

$$= \left(\prod_{i=1}^N \prod_{k=1}^K (\pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{\mathbb{I}(z_i=k)} \right) \times \quad (12.37)$$

$$\text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}) \prod_{k=1}^K \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_0, \mathbf{V}_0) \text{IW}(\boldsymbol{\Sigma}_k | \mathbf{S}_0, \nu_0) \quad (12.38)$$

We use the same prior for each mixture component.

The full conditionals are as follows. For the discrete indicators, we have

$$p(z_i = k | \mathbf{x}_i, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \propto \pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.39)$$

For the mixing weights, we have (using results from Section 3.4.2)

$$p(\boldsymbol{\pi}|\mathbf{z}) = \text{Dir}(\{\alpha_k + \sum_{i=1}^N \mathbb{I}(z_i = k)\}_{k=1}^K) \quad (12.40)$$

For the means, we have (using results from Section 3.4.4.1)

$$p(\boldsymbol{\mu}_k | \Sigma_k, \mathbf{z}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_k | \mathbf{m}_k, \mathbf{V}_k) \quad (12.41)$$

$$\mathbf{V}_k^{-1} = \mathbf{V}_0^{-1} + N_k \Sigma_k^{-1} \quad (12.42)$$

$$\mathbf{m}_k = \mathbf{V}_k (\Sigma_k^{-1} N_k \bar{\mathbf{x}}_k + \mathbf{V}_0^{-1} \mathbf{m}_0) \quad (12.43)$$

$$N_k \triangleq \sum_{i=1}^N \mathbb{I}(z_i = k) \quad (12.44)$$

$$\bar{\mathbf{x}}_k \triangleq \frac{\sum_{i=1}^N \mathbb{I}(z_i = k) \mathbf{x}_i}{N_k} \quad (12.45)$$

For the covariances, we have (using results from Section 3.4.4.2)

$$p(\Sigma_k | \boldsymbol{\mu}_k, \mathbf{z}, \mathbf{x}) = \text{IW}(\Sigma_k | \mathbf{S}_k, \nu_k) \quad (12.46)$$

$$\mathbf{S}_k = \mathbf{S}_0 + \sum_{i=1}^N \mathbb{I}(z_i = k) (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T \quad (12.47)$$

$$\nu_k = \nu_0 + N_k \quad (12.48)$$

12.3.6 Metropolis within Gibbs

When implementing Gibbs sampling, we have to sample from the full conditionals. If the distributions are conjugate, we can compute the full conditional in closed form, but in the general case, we will need to devise special algorithms to sample from the full conditionals.

One approach is to use the MH algorithm; this is called **Metropolis within Gibbs**. In particular, to sample from $x_i^{s+1} \sim p(x_i | \mathbf{x}_{1:i-1}^{s+1}, \mathbf{x}_{i+1:D}^s)$, we proceed in 3 steps:

1. Propose $x'_i \sim q(x'_i | x_i^s)$
2. Compute the acceptance probability $A_i = \min(1, \alpha_i)$ where

$$\alpha_i = \frac{p(\mathbf{x}_{1:i-1}^{s+1}, x'_i, \mathbf{x}_{i+1:D}^s) / q(x'_i | x_i^s)}{p(\mathbf{x}_{1:i-1}^s, x_i^s, \mathbf{x}_{i+1:D}^s) / q(x_i^s | x'_i)} \quad (12.49)$$

3. Sample $u \sim U(0, 1)$ and set $x_i^{s+1} = x'_i$ if $u < A_i$, and set $x_i^{s+1} = x_i^s$ otherwise.

12.3.7 Blocked Gibbs sampling

Gibbs sampling can be quite slow, since it only updates one variable at a time (so-called **single site updating**). If the variables are highly correlated, the chain will move slowly through the state space. This is illustrated in Figure 12.5, where we illustrate sampling from a 2d Gaussian. The ellipse represents the covariance matrix. The size of the moves taken by Gibbs sampling is controlled by the variance of the conditional distributions. If the variance is ℓ along some coordinate direction, but the support of the distribution is L along this dimension, then we need $O((L/\ell)^2)$ steps to obtain an independent sample.

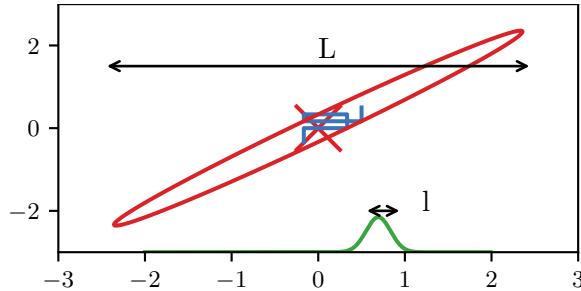


Figure 12.5: Illustration of potentially slow sampling when using Gibbs sampling for a skewed 2d Gaussian. Adapted from Figure 11.11 of [Bis06]. Generated by `gibbs_gauss_demo.ipynb`.

In some cases we can efficiently sample groups of variables at a time. This is called **blocked Gibbs sampling** [JKK95; WY02], and can make much bigger moves through the state space.

As an example, suppose we want to perform Bayesian inference for a state-space model, such as an HMM, i.e., we want to sample from

$$p(\boldsymbol{\theta}, \mathbf{z} | \mathbf{x}) \propto p(\boldsymbol{\theta}) \prod_{t=1}^T p(\mathbf{x}_t | \mathbf{z}_t, \boldsymbol{\theta}) p(\mathbf{z}_t | \mathbf{z}_{t-1}, \boldsymbol{\theta}) \quad (12.50)$$

We can use blocked Gibbs sampling, where we alternate between sampling from $p(\boldsymbol{\theta} | \mathbf{z}, \mathbf{x})$ and $p(\mathbf{z} | \mathbf{x}, \boldsymbol{\theta})$. The former is easy to do (assuming conjugate priors), since all variables in the model are observed (see Section 29.8.4.1). The latter can be done using forwards-filtering backwards-sampling (Section 9.2.7).

12.3.8 Collapsed Gibbs sampling

We can sometimes gain even greater speedups by analytically integrating out some of the unknown quantities. This is called a **collapsed Gibbs sampler**, and it tends to be more efficient, since it is sampling in a lower dimensional space. This can result in lower variance, as discussed in Section 11.6.2.

As an example, consider a GMM with a fully conjugate prior. This can be represented as a DPGM as shown in Figure 12.6a. Since the prior is conjugate, we can analytically integrate out the model parameters $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, and $\boldsymbol{\pi}$, so the only remaining hidden variables are the discrete indicator variables \mathbf{z} . However, once we integrate out $\boldsymbol{\pi}$, all the z_i nodes become inter-dependent. Similarly, once we integrate out $\boldsymbol{\theta}_k = (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, all the \mathbf{x}_i nodes become inter-dependent, as shown in Figure 12.6b. Nevertheless, we can easily compute the full conditionals, and hence implement a Gibbs sampler, as we explain below. In particular, the full conditional for the latent indicators is given by

$$p(z_i = k | \mathbf{z}_{-i}, \mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}, \boldsymbol{\beta}) p(\mathbf{x} | z_i = k, \mathbf{z}_{-i}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \quad (12.51)$$

$$\propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}) p(\mathbf{x}_i | \mathbf{x}_{-i}, z_i = k, \mathbf{z}_{-i}, \boldsymbol{\beta}) \quad (12.52)$$

$$\propto p(z_i = k | \mathbf{z}_{-i}, \boldsymbol{\alpha}) p(\mathbf{x}_i | \mathbf{x}_{-i}, z_i = k, \mathbf{z}_{-i}, \boldsymbol{\beta}) \quad (12.53)$$

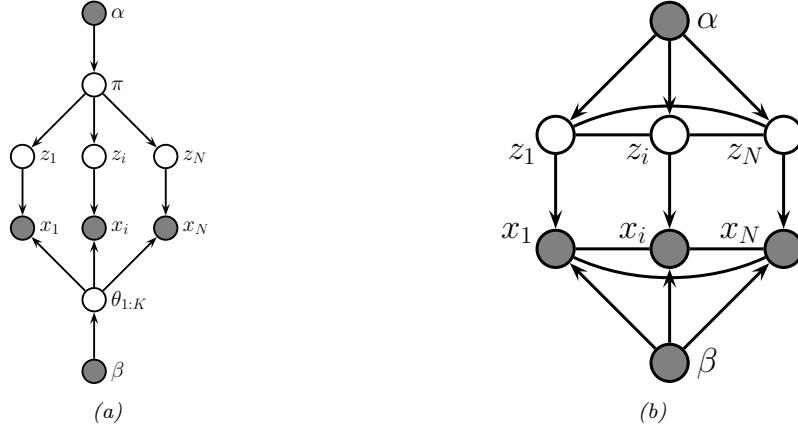


Figure 12.6: (a) A mixture model represented as an “unrolled” DPGM. (b) After integrating out the continuous latent parameters.

where $\boldsymbol{\beta} = (\mathbf{m}_0, \mathbf{V}_0, \mathbf{S}_0, \nu_0)$ are the hyper-parameters for the class-conditional densities. We now discuss how to compute these terms.

Suppose we use a symmetric prior of the form $\boldsymbol{\pi} \sim \text{Dir}(\boldsymbol{\alpha})$, where $\alpha_k = \alpha/K$, for the mixing weights. Then we can obtain the first term in Equation (12.53), from Equation (3.96), where

$$p(z_1, \dots, z_N | \alpha) = \frac{\Gamma(\alpha)}{\Gamma(N + \alpha)} \prod_{k=1}^K \frac{\Gamma(N_k + \alpha/K)}{\Gamma(\alpha/K)} \quad (12.54)$$

Hence

$$p(z_i = k | \mathbf{z}_{-i}, \alpha) = \frac{p(\mathbf{z}_{1:N} | \alpha)}{p(\mathbf{z}_{-i} | \alpha)} = \frac{\frac{1}{\Gamma(N+\alpha)}}{\frac{1}{\Gamma(N+\alpha-1)}} \times \frac{\Gamma(N_k + \alpha/K)}{\Gamma(N_{k,-i} + \alpha/K)} \quad (12.55)$$

$$= \frac{\Gamma(N + \alpha - 1)}{\Gamma(N + \alpha)} \frac{\Gamma(N_{k,-i} + 1 + \alpha/K)}{\Gamma(N_{k,-i} + \alpha/K)} = \frac{N_{k,-i} + \alpha}{N + \alpha - 1} \quad (12.56)$$

where $N_{k,-i} \triangleq \sum_{n \neq i} \mathbb{I}(z_n = k) = N_k - 1$, and where we exploited the fact that $\Gamma(x+1) = x\Gamma(x)$.

To obtain the second term in Equation (12.53), which is the posterior predictive distribution for \mathbf{x}_i given all the other data and all the assignments, we use the fact that

$$p(\mathbf{x}_i | \mathbf{x}_{-i}, \mathbf{z}_{-i}, z_i = k, \boldsymbol{\beta}) = p(\mathbf{x}_i | \mathcal{D}_{-i,k}, \boldsymbol{\beta}) \quad (12.57)$$

where $\mathcal{D}_{-i,k} = \{\mathbf{x}_j : z_j = k, j \neq i\}$ is all the data assigned to cluster k except for \mathbf{x}_i . If we use a conjugate prior for $\boldsymbol{\theta}_k$, we can compute $p(\mathbf{x}_i | \mathcal{D}_{-i,k}, \boldsymbol{\beta})$ in closed form. Furthermore, we can efficiently update these predictive likelihoods by caching the sufficient statistics for each cluster. To compute the above expression, we remove \mathbf{x}_i ’s statistics from its current cluster (namely z_i), and then evaluate \mathbf{x}_i under each cluster’s posterior predictive distribution. Once we have picked a new cluster, we add \mathbf{x}_i ’s statistics to this new cluster.

Some pseudo-code for one step of the algorithm is shown in Algorithm 12.2, based on [Sud06, p94]. (We update the nodes in random order to improve the mixing time, as suggested in [RS97b].) We can initialize the sample by sequentially sampling from $p(z_i | \mathbf{z}_{1:i-1}, \mathbf{x}_{1:i})$. In the case of GMMs, both the naive sampler and collapsed sampler take $O(NKD)$ time per step.

Algorithm 12.2: Collapsed Gibbs sampler for a mixture model

```

1 for each  $i = 1 : N$  in random order do
2   Remove  $\mathbf{x}_i$ 's sufficient statistics from old cluster  $z_i$ 
3   for each  $k = 1 : K$  do
4      $\lfloor$  Compute  $p_k(\mathbf{x}_i | \boldsymbol{\beta}) = p(\mathbf{x}_i | \{\mathbf{x}_j : z_j = k, j \neq i\}, \boldsymbol{\beta})$ 
5     Compute  $p(z_i = k | \mathbf{z}_{-i}, \alpha) \propto (N_{k,-i} + \alpha/K)p_k(\mathbf{x}_i)$ 
6     Sample  $z_i \sim p(z_i | \cdot)$ 
7    $\rfloor$  Add  $\mathbf{x}_i$ 's sufficient statistics to new cluster  $z_i$ 

```

The primary advantage of using the collapsed sampler is that it extends to the case where we have an “infinite” number of mixture components, as in the Dirichlet process mixture model of Supplementary Section 31.2.2.

12.4 Auxiliary variable MCMC

Sometimes we can dramatically improve the efficiency of sampling by introducing **auxiliary variables**, in order to reduce correlation between the original variables. If the original variables are denoted by \mathbf{x} , and the auxiliary variables by \mathbf{v} , then the augmented distribution becomes $p(\mathbf{x}, \mathbf{v})$. We assume it is easier to sample from this than the marginal distribution $p(\mathbf{x})$. If so, we can draw joint samples $(\mathbf{x}^s, \mathbf{v}^s) \sim p(\mathbf{x}, \mathbf{v})$, and then just “throw away” the \mathbf{v}^s , and the result will be samples from the desired marginal, $\mathbf{x}^s \sim \sum_{\mathbf{v}} p(\mathbf{x}, \mathbf{v})$. We give some examples of this below.

12.4.1 Slice sampling

Consider sampling from a univariate, but multimodal, distribution $p(x) = \tilde{p}(x)/Z_p$, where $\tilde{p}(x)$ is unnormalized, and $Z_p = \int \tilde{p}(x)dx$. We can sometimes improve the ability to make large moves by adding a uniform auxiliary variable v . We define the joint distribution as follows:

$$\hat{p}(x, v) = \begin{cases} 1/Z_p & \text{if } 0 \leq v \leq \tilde{p}(x) \\ 0 & \text{otherwise} \end{cases} \quad (12.58)$$

The marginal distribution over x is given by

$$\int \hat{p}(x, v) dv = \int_0^{\tilde{p}(x)} \frac{1}{Z_p} dv = \frac{\tilde{p}(x)}{Z_p} = p(x) \quad (12.59)$$

so we can sample from $p(x)$ by sampling from $\hat{p}(x, v)$ and then ignoring v . To do this, we will use a technique called **slice sampling** [Nea03].

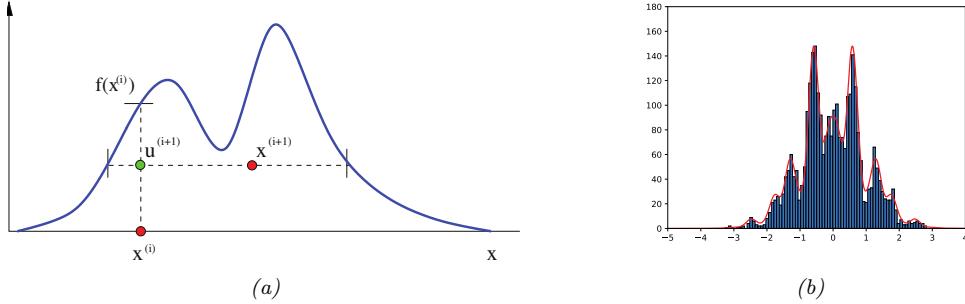


Figure 12.7: Slice sampling. (a) Illustration of one step of the algorithm in 1d. Given a previous sample x^i , we sample u^{i+1} uniformly on $[0, f(x^i)]$, where $f = \tilde{p}$ is the (unnormalized) target density. We then sample x^{i+1} along the slice where $f(x) \geq u^{i+1}$. From Figure 15 of [And+03]. Used with kind permission of Nando de Freitas. (b) Output of slice sampling applied to a 1d distribution. Generated by `slice_sampling_demo_1d.ipynb`.

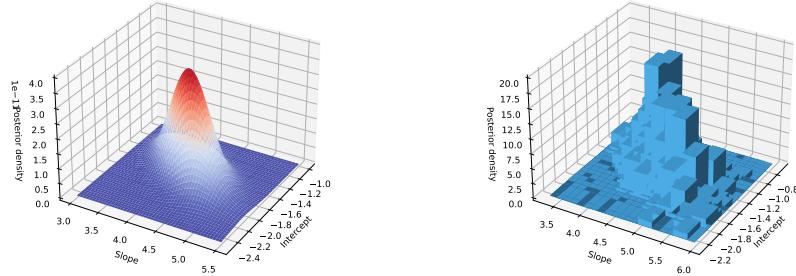


Figure 12.8: Posterior for binomial regression for 1d data. Left: slice sampling approximation. Right: grid approximation. Generated by `slice_sampling_demo_2d.ipynb`.

This works as follows. Given previous sample x^i , we sample v^{i+1} from

$$p(v|x^i) = U_{[0,\tilde{p}(x^i)]}(v) \quad (12.60)$$

This amounts to uniformly picking a point on the vertical line between 0 and $\tilde{p}(x^i)$. We use this to construct a “slice” of the density at or above this height, by computing $A^{i+1} = \{x : \tilde{p}(x) \geq v^{i+1}\}$. We then sample x^{i+1} uniformly from this set. See Figure 12.7(a) for an illustration.

To compute the level set A , we can use an iterative search procedure called **stepping out**, in which we start with an interval $x_{min} \leq x \leq x_{max}$ around the current point x^i of some width, and then we keep extending it until the endpoints fall outside the slice. We can then use rejection sampling to sample from the interval. For the details, see [Nea03].

To apply the method to multivariate distributions, we sample one extra auxiliary variable for each dimension. Thus we perform 2D sampling operations to draw a single joint sample, where

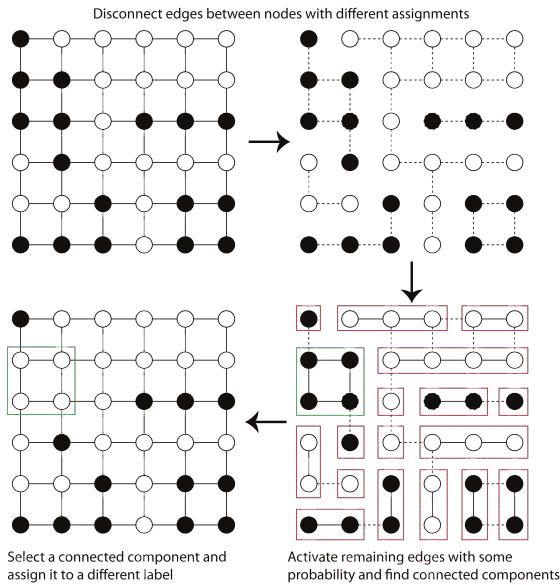


Figure 12.9: Illustration of the Swendsen-Wang algorithm on a 2d grid. Used with kind permission of Kevin Tang.

D is the number of random variables. The advantage of this over Gibbs sampling applied to the original (non-augmented) distribution is that it only needs access to the unnormalized joint, not the full-conditionals.

Figure 12.7(b) illustrates the algorithm in action on a synthetic 1d problem. Figure 12.8 illustrates its behavior on a slightly harder problem, namely binomial logistic regression. The model has the form $y_i \sim \text{Bin}(n_i, \text{logit}(\beta_1 + \beta_2 x_i))$. We use a vague Gaussian prior for the β_j 's. On the left we show the slice sampling approximation to the posterior, and on the right we show a grid-based approximation, as a simple deterministic proxy for the true posterior. We see a close correspondence.

12.4.2 Swendsen-Wang

Consider an Ising model of the following form: $p(\mathbf{x}) = \frac{1}{Z} \prod_e \Psi(\mathbf{x}_e)$, where $\mathbf{x}_e = (x_i, x_j)$ for edge $e = (i, j)$, $x_i \in \{+1, -1\}$, and the edge potential is defined by $\begin{pmatrix} e^J & e^{-J} \\ e^{-J} & e^J \end{pmatrix}$, where J is the edge strength. In Section 12.3.3, we discussed how to apply Gibbs sampling to this model. However, this can be slow when J is large in absolute value, because neighboring states can be highly correlated. The **Swendsen-Wang** algorithm [SW87b] is an auxiliary variable MCMC sampler which mixes much faster, at least for the case of attractive or ferromagnetic models, with $J > 0$.

Suppose we introduce auxiliary binary variables, one per edge.⁴ These are called **bond variables**, and will be denoted by \mathbf{v} . We then define an extended model $p(\mathbf{x}, \mathbf{v})$ of the form $p(\mathbf{x}, \mathbf{v}) =$

4. Our presentation of the method is based on notes by David MacKay, available from <http://www.inference.phy.cam.ac.uk/mackay/itila/swendsen.pdf>.

$\frac{1}{Z'} \prod_e \Psi(\mathbf{x}_e, v_e)$, where $v_e \in \{0, 1\}$, and we define the new edge potentials as follows:

$$\Psi(\mathbf{x}_e, v_e = 0) = \begin{pmatrix} e^{-J} & e^{-J} \\ e^{-J} & e^{-J} \end{pmatrix}, \quad \Psi(\mathbf{x}_e, v_e = 1) = \begin{pmatrix} e^J - e^{-J} & 0 \\ 0 & e^J - e^{-J} \end{pmatrix} \quad (12.61)$$

It is clear that $\sum_{v_e=0}^1 \Psi(\mathbf{x}_e, v_e) = \Psi(\mathbf{x}_e)$, and hence that $\sum_{\mathbf{v}} p(\mathbf{x}, \mathbf{v}) = p(\mathbf{x})$, as required.

Fortunately, it is easy to apply Gibbs sampling to this extended model. The full conditional $p(\mathbf{v}|\mathbf{x})$ factorizes over the edges, since the bond variables are conditionally independent given the node variables. Furthermore, the full conditional $p(v_e|\mathbf{x}_e)$ is simple to compute: if the nodes on either end of the edge are in the same state ($x_i = x_j$), we set the bond v_e to 1 with probability $p = 1 - e^{-2J}$, otherwise we set it to 0. In Figure 12.9 (top right), the bonds that could be turned on (because their corresponding nodes are in the same state) are represented by dotted edges. In Figure 12.9 (bottom right), the bonds that are randomly turned on are represented by solid edges.

To sample $p(\mathbf{x}|\mathbf{v})$, we proceed as follows. Find the connected components defined by the graph induced by the bonds that are turned on. (Note that a connected component may consist of a singleton node.) Pick one of these components uniformly at random. All the nodes in each such component must have the same state. Pick a state ± 1 uniformly at random, and set all the variables in this component to adopt this new state. This is illustrated in Figure 12.9 (bottom right), where the green square denotes the selected connected component; we set all the nodes within this square to white, to get the bottom left configuration.

It should be intuitively clear that Swendsen-Wang makes much larger moves through the state space than Gibbs sampling. The gains are exponentially large for certain settings of the edge parameter. More precisely, let the edge strength be parameterized by J/T , where $T > 0$ is a computational temperature. For large T , the nodes are roughly independent, so both methods work equally well. However, as T approaches a **critical temperature** T_c , the typical states of the system have very long correlation lengths, and Gibbs sampling takes a very long time to generate independent samples. As the temperature continues to drop, the typical states are either all on or all off. The frequency with which Gibbs sampling moves between these two modes is exponentially small. By contrast, SW mixes rapidly at all temperatures.

Unfortunately, if any of the edge weights are negative, $J < 0$, the system is **frustrated**, and there are exponentially many modes, even at low temperature. SW does not work very well in this setting, since it tries to force many neighboring variables to have the same state. In fact, sampling from these kinds of frustrated systems is provably computationally hard for any algorithm [JS93; JS96].

12.5 Hamiltonian Monte Carlo (HMC)

Many MCMC algorithms perform poorly in high dimensional spaces, because they rely on a form of random search based on local perturbations. In this section, we discuss a method known as **Hamiltonian Monte Carlo** or **HMC**, that leverages gradient information to guide the local moves. This is an auxiliary variable method (Section 12.4) derived from physics [Dua+87; Nea93; Mac03; Nea10; Bet17].⁵ In particular, the method builds on **Hamiltonian mechanics**, which we describe below.

5. The method was originally called **hybrid MC** [Dua+87]. It was introduced to the statistics community in [Nea93], and was renamed to Hamiltonian MC in [Mac03].

12.5.1 Hamiltonian mechanics

Consider a particle rolling around an energy landscape. We can characterize the motion of the particle in terms of its position $\boldsymbol{\theta} \in \mathbb{R}^D$ (often denoted by \mathbf{q}) and its momentum $\mathbf{v} \in \mathbb{R}^D$ (often denoted by \mathbf{p}). The set of possible values for $(\boldsymbol{\theta}, \mathbf{v})$ is called the **phase space**. We define the **Hamiltonian** function for each point in phase space as follows:

$$\mathcal{H}(\boldsymbol{\theta}, \mathbf{v}) \triangleq \mathcal{E}(\boldsymbol{\theta}) + \mathcal{K}(\mathbf{v}) \quad (12.62)$$

where $\mathcal{E}(\boldsymbol{\theta})$ is the **potential energy**, $\mathcal{K}(\mathbf{v})$ is the **kinetic energy**, and the Hamiltonian is the total energy. In a physical setting, the potential energy is due to the pull of gravity, and the momentum is due to the motion of the particle. In a statistical setting, we often take the potential energy to be

$$\mathcal{E}(\boldsymbol{\theta}) = -\log \tilde{p}(\boldsymbol{\theta}) \quad (12.63)$$

where $\tilde{p}(\boldsymbol{\theta})$ is a possibly unnormalized distribution, such as $p(\boldsymbol{\theta}, \mathcal{D})$, and the kinetic energy to be

$$\mathcal{K}(\mathbf{v}) = \frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma}^{-1} \mathbf{v} \quad (12.64)$$

where $\boldsymbol{\Sigma}$ is a positive definite matrix, known as the **inverse mass matrix**.

Stable orbits are defined by trajectories in phase space that have a constant energy. The trajectory of a particle within an energy level set can be obtained by solving the following continuous time differential equations, known as **Hamilton's equations**:

$$\begin{aligned} \frac{d\boldsymbol{\theta}}{dt} &= \frac{\partial \mathcal{H}}{\partial \mathbf{v}} = \frac{\partial \mathcal{K}}{\partial \mathbf{v}} \\ \frac{d\mathbf{v}}{dt} &= -\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}} = -\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} \end{aligned} \quad (12.65)$$

To see why energy is conserved, note that

$$\frac{d\mathcal{H}}{dt} = \sum_{i=1}^D \left[\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{d\boldsymbol{\theta}_i}{dt} + \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} \frac{d\mathbf{v}_i}{dt} \right] = \sum_{i=1}^D \left[\frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} - \frac{\partial \mathcal{H}}{\partial \boldsymbol{\theta}_i} \frac{\partial \mathcal{H}}{\partial \mathbf{v}_i} \right] = 0 \quad (12.66)$$

Intuitively, we can understand this result as follows: a satellite in orbit around a planet will “want” to continue in a straight line due to its momentum, but will get pulled in towards the planet due to gravity, and if these forces cancel, the orbit is stable. If the satellite starts spiraling towards the planet, its kinetic energy will increase but its potential energy will decrease.

Note that the mapping from $(\boldsymbol{\theta}(t), \mathbf{v}(t))$ to $(\boldsymbol{\theta}(t+s), \mathbf{v}(t+s))$ for some time increment s is invertible for small enough time steps. Furthermore, this mapping is volume preserving, so has a Jacobian determinant of 1. (See e.g., [BZ20, p287] for a proof.) These facts will be important later when we turn this system into an MCMC algorithm.

12.5.2 Integrating Hamilton's equations

In this section, we discuss how to simulate Hamilton's equations in discrete time.

12.5.2.1 Euler's method

The simplest way to model the time evolution is to update the position and momentum simultaneously by a small amount, known as the step size η :

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \eta \frac{d\mathbf{v}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_t) = \mathbf{v}(t) - \eta \frac{\partial \mathcal{E}(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}} \quad (12.67)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \frac{d\boldsymbol{\theta}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_t) = \boldsymbol{\theta}_t + \eta \frac{\partial \mathcal{K}(\mathbf{v}_t)}{\partial \mathbf{v}} \quad (12.68)$$

If the kinetic energy has the form in Equation (12.64) then the second expression simplifies to

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \boldsymbol{\Sigma}^{-1} \mathbf{v}_{t+1} \quad (12.69)$$

This is known as **Euler's method**.

12.5.2.2 Modified Euler's method

The **modified Euler's method** is slightly more accurate, and works as follows: First update the momentum, and then update the position using the new momentum:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \eta \frac{d\mathbf{v}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_t) = \mathbf{v}_t - \eta \frac{\partial \mathcal{E}(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}} \quad (12.70)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \frac{d\boldsymbol{\theta}}{dt}(\boldsymbol{\theta}_t, \mathbf{v}_{t+1}) = \boldsymbol{\theta}_t + \eta \frac{\partial \mathcal{K}(\mathbf{v}_{t+1})}{\partial \mathbf{v}} \quad (12.71)$$

Unfortunately, the asymmetry of this method can cause some theoretical problems (see e.g., [BZ20, p287]) which we resolve below.

12.5.2.3 Leapfrog integrator

In this section, we discuss the **leapfrog integrator**, which is a symmetrized version of the modified Euler method. We first perform a “half” update of the momentum, then a full update of the position, and then finally another “half” update of the momentum:

$$\mathbf{v}_{t+1/2} = \mathbf{v}_t - \frac{\eta}{2} \frac{\partial \mathcal{E}(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}} \quad (12.72)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \frac{\partial \mathcal{K}(\mathbf{v}_{t+1/2})}{\partial \mathbf{v}} \quad (12.73)$$

$$\mathbf{v}_{t+1} = \mathbf{v}_{t+1/2} - \frac{\eta}{2} \frac{\partial \mathcal{E}(\boldsymbol{\theta}_{t+1})}{\partial \boldsymbol{\theta}} \quad (12.74)$$

If we perform multiple leapfrog steps, it is equivalent to performing a half step update of \mathbf{v} at the beginning and end of the trajectory, and alternating between full step updates of $\boldsymbol{\theta}$ and \mathbf{v} in between.

12.5.2.4 Higher order integrators

It is possible to define higher order integrators that are more accurate, but take more steps. For details, see [BRSS18].

12.5.3 The HMC algorithm

We now describe how to use Hamiltonian dynamics to define an MCMC sampler in the expanded state space $(\boldsymbol{\theta}, \mathbf{v})$. The target distribution has the form

$$p(\boldsymbol{\theta}, \mathbf{v}) = \frac{1}{Z} \exp[-\mathcal{H}(\boldsymbol{\theta}, \mathbf{v})] = \frac{1}{Z} \exp \left[-\mathcal{E}(\boldsymbol{\theta}) - \frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v} \right] \quad (12.75)$$

The marginal distribution over the latent variables of interest has the form

$$p(\boldsymbol{\theta}) = \int p(\boldsymbol{\theta}, \mathbf{v}) d\mathbf{v} = \frac{1}{Z_q} e^{-\mathcal{E}(\boldsymbol{\theta})} \int \frac{1}{Z_p} e^{-\frac{1}{2} \mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v}} d\mathbf{v} = \frac{1}{Z_q} e^{-\mathcal{E}(\boldsymbol{\theta})} \quad (12.76)$$

Suppose the previous state of the Markov chain is $(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$. To sample the next state, we proceed as follows. We set the initial position to $\boldsymbol{\theta}'_0 = \boldsymbol{\theta}_{t-1}$, and sample a new random momentum, $\mathbf{v}'_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. We then initialize a random trajectory in the phase space, starting at $(\boldsymbol{\theta}'_0, \mathbf{v}'_0)$, and followed for L leapfrog steps, until we get to the final proposed state $(\boldsymbol{\theta}^*, \mathbf{v}^*) = (\boldsymbol{\theta}'_L, \mathbf{v}'_L)$. If we have simulated Hamiltonian mechanics correctly, the energy should be the same at the start and end of this process; if not, we say the HMC has **diverged**, and we reject the sample. If the energy is constant, we compute the MH acceptance probability

$$\alpha = \min \left(1, \frac{p(\boldsymbol{\theta}^*, \mathbf{v}^*)}{p(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})} \right) = \min (1, \exp[-\mathcal{H}(\boldsymbol{\theta}^*, \mathbf{v}^*) + \mathcal{H}(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})]) \quad (12.77)$$

(The transition probabilities cancel since the proposal is reversible.) Finally, we accept the proposal by setting $(\boldsymbol{\theta}_t, \mathbf{v}_t) = (\boldsymbol{\theta}^*, \mathbf{v}^*)$ with probability α , otherwise we set $(\boldsymbol{\theta}_t, \mathbf{v}_t) = (\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$. (In practice we don't need to keep the momentum term, it is only used inside of the leapfrog algorithm.) See Algorithm 12.3 for the pseudocode.⁶

Algorithm 12.3: Hamiltonian Monte Carlo

```

1 for  $t = 1 : T$  do
2   Generate random momentum  $\mathbf{v}_{t-1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ 
3   Set  $(\boldsymbol{\theta}'_0, \mathbf{v}'_0) = (\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})$ 
4   Half step for momentum:  $\mathbf{v}'_{\frac{1}{2}} = \mathbf{v}'_0 - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}'_0)$ 
5   for  $l = 1 : L - 1$  do
6      $\boldsymbol{\theta}'_l = \boldsymbol{\theta}'_{l-1} + \eta \boldsymbol{\Sigma}^{-1} \mathbf{v}'_{l-1/2}$ 
7      $\mathbf{v}'_{l+1/2} = \mathbf{v}'_{l-1/2} - \eta \nabla \mathcal{E}(\boldsymbol{\theta}'_l)$ 
8   Full step for location:  $\boldsymbol{\theta}'_L = \boldsymbol{\theta}'_{L-1} + \eta \boldsymbol{\Sigma}^{-1} \mathbf{v}'_{L-1/2}$ 
9   Half step for momentum:  $\mathbf{v}'_L = \mathbf{v}'_{L-1/2} - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}'_L)$ 
10  Compute proposal  $(\boldsymbol{\theta}^*, \mathbf{v}^*) = (\boldsymbol{\theta}'_L, \mathbf{v}'_L)$ 
11  Compute  $\alpha = \min (1, \exp[-\mathcal{H}(\boldsymbol{\theta}^*, \mathbf{v}^*) + \mathcal{H}(\boldsymbol{\theta}_{t-1}, \mathbf{v}_{t-1})])$ 
12  Set  $\boldsymbol{\theta}_t = \boldsymbol{\theta}^*$  with probability  $\alpha$ , otherwise  $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1}$ .

```

6. There are many high-quality implementations of HMC. For example, [BlackJAX](#) in JAX.

We need to sample a new momentum at each iteration to satisfy ergodicity. To see why, recall that $\mathcal{H}(\boldsymbol{\theta}, \mathbf{v})$ stays approximately constant as we move through phase space. If $\mathcal{H}(\boldsymbol{\theta}, \mathbf{v}) = \mathcal{E}(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{v}^T \boldsymbol{\Sigma} \mathbf{v}$, then clearly $\mathcal{E}(\boldsymbol{\theta}) \leq \mathcal{H}(\boldsymbol{\theta}, \mathbf{v}) = h$ for all locations $\boldsymbol{\theta}$ along the trajectory. Thus the sampler cannot reach states where $\mathcal{E}(\boldsymbol{\theta}) > h$. To ensure the sampler explores the full space, we must pick a random momentum at the start of each iteration.

12.5.4 Tuning HMC

We need to specify three hyperparameters for HMC: the number of leapfrog steps L , the step size η , and the covariance $\boldsymbol{\Sigma}$.

12.5.4.1 Choosing the number of steps using NUTS

We want to choose the number of leapfrog steps L to be large enough that the algorithm explores the level set of constant energy, but without doubling back on itself, which would waste computation, due to correlated samples. Fortunately, there is an algorithm, known as the **no-U-turn sampler** or **NUTS** algorithm [HG14], which can adaptively choose L for us.

12.5.4.2 Choosing the step size

When $\boldsymbol{\Sigma} = \mathbf{I}$, the ideal step size η should be roughly equal to the width of $\mathcal{E}(\boldsymbol{\theta})$ in the most constrained direction of the local energy landscape. For a locally quadratic potential, this corresponds to the square root of the smallest marginal standard deviation of the local covariance matrix. (If we think of the energy surface as a valley, this corresponds to the direction with the steepest sides.) A step size much larger than this will cause moves that are likely to be rejected because they move to places which increase the potential energy too much. On the other hand, if the step size is too low, the proposal distribution will not move much from the starting position, and the algorithm will be very slow.

In [BZ20, Sec 9.5.4] they recommend the following heuristic for picking η : set $\boldsymbol{\Sigma} = \mathbf{I}$ and $L = 1$, and then vary η until the acceptance rates are in the range of 40%–80%. Of course, different step sizes might be needed in different parts of the state space. In this case, we can use learning rate schedules from the optimization literature, such as cyclical schedules [Zha+20d].

12.5.4.3 Choosing the covariance (inverse mass) matrix

To allow for larger step sizes, we can use a smarter choice for $\boldsymbol{\Sigma}$, also called the **inverse mass matrix**. One way to estimate a fixed $\boldsymbol{\Sigma}$ is to run HMC with $\boldsymbol{\Sigma} = \mathbf{I}$ for a **warm-up** period, until the chain is “burned in” (see Section 12.6); then we run for a few more steps, so we can compute the empirical covariance matrix using $\boldsymbol{\Sigma} = \mathbb{E}[(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})(\boldsymbol{\theta} - \bar{\boldsymbol{\theta}})^T]$. In [Hof+19] they propose a method called the **NeuTra HMC** algorithm which “neutralizes” bad geometry by learning an inverse autoregressive flow model (Section 23.2.4.3) in order to map the warped distribution to an isotropic Gaussian. This is often an order of magnitude faster than vanilla HMC.

12.5.5 Riemann manifold HMC

If we let the covariance matrix change as we move position, so Σ is a function of θ , the method is known as **Riemann manifold HMC** or **RM-HMC** [GC11; Bet13], since the moves follow a curved manifold, rather than the flat manifold induced by a constant Σ .

A natural choice for the covariance matrix is to use the Hessian at the current location, to capture the local geometry:

$$\Sigma(\theta) = \nabla^2 \mathcal{E}(\theta) \quad (12.78)$$

Since this is not always positive definite, an alternative, that can be used for some problems, is to use the Fisher information matrix (Section 3.3.4), given by

$$\Sigma(\theta) = -\mathbb{E}_{p(x|\theta)} [\nabla^2 \log p(x|\theta)] \quad (12.79)$$

Once we have computed $\Sigma(\theta)$, we can compute the kinetic energy as follows:

$$\mathcal{K}(\theta, v) = \frac{1}{2} \log((2\pi)^D |\Sigma(\theta)|) + \frac{1}{2} v^\top \Sigma(\theta) v \quad (12.80)$$

Unfortunately the Hamiltonian updates of θ and v are no longer separable, which makes the RM-HMC algorithm more complex to implement, so it is not widely used.

12.5.6 Langevin Monte Carlo (MALA)

A special case of HMC occurs when we take $L = 1$ leapfrog steps. This is known as **Langevin Monte Carlo (LMC)**, or the **Metropolis adjusted Langevin algorithm (MALA)** [RT96]. This gives rise to the simplified algorithm shown in Algorithm 12.4.

Algorithm 12.4: Langevin Monte Carlo

```

1 for t = 1 : T do
2   Generate random momentum  $v_{t-1} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ 
3    $\theta^* = \theta_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\theta_{t-1}) + \eta \Sigma^{-1} v_{t-1}$ 
4    $v^* = v_{t-1} - \frac{\eta}{2} \nabla \mathcal{E}(\theta_{t-1}) - \frac{\eta}{2} \nabla \mathcal{E}(\theta^*)$ 
5   Compute  $\alpha = \min(1, \exp[-\mathcal{H}(\theta^*, v^*)] / \exp[-\mathcal{H}(\theta_{t-1}, v_{t-1})])$ 
6   Set  $\theta_t = \theta^*$  with probability  $\alpha$ , otherwise  $\theta_t = \theta_{t-1}$ .

```

A further simplification is to eliminate the MH acceptance step. In this case, the update becomes

$$\theta_t = \theta_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\theta_{t-1}) + \eta \Sigma^{-1} v_{t-1} \quad (12.81)$$

$$= \theta_{t-1} - \frac{\eta^2}{2} \Sigma^{-1} \nabla \mathcal{E}(\theta_{t-1}) + \eta \sqrt{\Sigma^{-1}} \epsilon_{t-1} \quad (12.82)$$

where $v_{t-1} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ and $\epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. This is just like gradient descent with added noise. If we set Σ to be the Fisher information matrix, this becomes natural gradient descent (Section 6.4) with

added noise. If we approximate the gradient with a stochastic gradient, we get a method known as **SGLD**, or **stochastic gradient Langevin descent** (see Section 12.7.1 for details).

Now suppose $\Sigma = \mathbf{I}$, and we set $\eta = \sqrt{2}$. In continuous time, we get the following stochastic differential equation (SDE), known as **Langevin diffusion**:

$$d\boldsymbol{\theta}_t = -\nabla \mathcal{E}(\boldsymbol{\theta}_t) dt + \sqrt{2} d\mathbf{B}_t \quad (12.83)$$

where \mathbf{B}_t represents D -dimensional **Brownian motion**. If we use this to generate the samples, the method is known as the **anadjusted Langevin algorithm** or **ULA** [Par81; RT96].

12.5.7 Connection between SGD and Langevin sampling

In this section, we discuss a deep connection between stochastic gradient descent (SGD) and Langevin sampling, following the presentation of [BZ20, Sec 10.2.3].

Consider the minimization of the additive loss

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^N \mathcal{L}_n(\boldsymbol{\theta}) \quad (12.84)$$

For example, we may define $\mathcal{L}_n(\boldsymbol{\theta}) = -\log p(y_n | \mathbf{x}_n, \boldsymbol{\theta})$. We will use a minibatch approximation to the gradients:

$$\nabla_B \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{n \in \mathcal{S}} \nabla \mathcal{L}_n(\boldsymbol{\theta}) \quad (12.85)$$

where $\mathcal{S} = \{i_1, \dots, i_B\}$ is a randomly chosen set of indices of size B . For simplicity of analysis, we assume the indices are chosen with replacement from $\{1, \dots, N\}$.

Let us define the (scaled) error (due to minibatching) in the estimated gradient by

$$\mathbf{v}_t \triangleq \sqrt{\eta} (\nabla \mathcal{L}(\boldsymbol{\theta}_t) - \nabla_B \mathcal{L}(\boldsymbol{\theta}_t)) \quad (12.86)$$

This is called the **diffusion term**. Then we can rewrite the SGD update as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_B \mathcal{L}(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t) + \sqrt{\eta} \mathbf{v}_t \quad (12.87)$$

The diffusion term \mathbf{v}_t has mean 0, since

$$\mathbb{E} [\nabla_B \mathcal{L}(\boldsymbol{\theta})] = \frac{1}{B} \sum_{j=1}^B \mathbb{E} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})] = \frac{1}{B} \sum_{j=1}^B \nabla \mathcal{L}(\boldsymbol{\theta}) = \nabla \mathcal{L}(\boldsymbol{\theta}) \quad (12.88)$$

To compute the variance of the diffusion term, note that

$$\mathbb{V} [\nabla_B \mathcal{L}(\boldsymbol{\theta})] = \frac{1}{B^2} \sum_{j=1}^B \mathbb{V} [\nabla \mathcal{L}_{i_j}(\boldsymbol{\theta})] \quad (12.89)$$

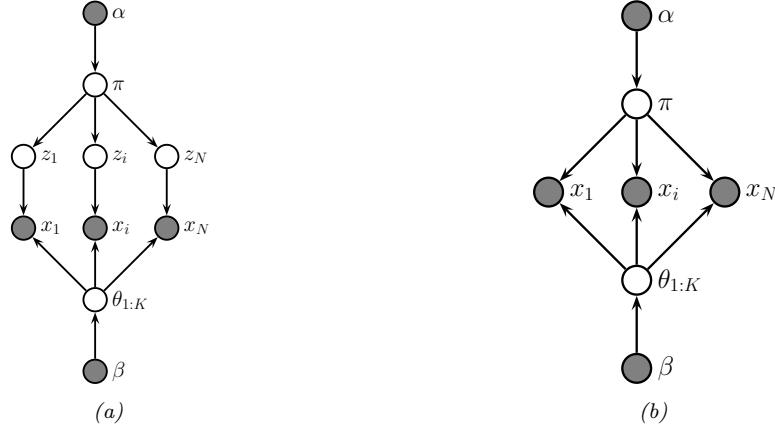


Figure 12.10: (a) A mixture model. (b) After integrating out the discrete latent variables.

where

$$\mathbb{V} [\nabla \mathcal{L}_{i,j}(\boldsymbol{\theta})] = \mathbb{E} [\nabla \mathcal{L}_{i,j}(\boldsymbol{\theta}) \nabla \mathcal{L}_{i,j}(\boldsymbol{\theta})^\top] - \mathbb{E} [\nabla \mathcal{L}_{i,j}(\boldsymbol{\theta})] \mathbb{E} [\nabla \mathcal{L}_{i,j}(\boldsymbol{\theta})^\top] \quad (12.90)$$

$$= \left(\frac{1}{N} \sum_{n=1}^N \nabla \mathcal{L}_n(\boldsymbol{\theta}) \nabla \mathcal{L}_n(\boldsymbol{\theta})^\top \right) - \nabla \mathcal{L}(\boldsymbol{\theta}) \nabla \mathcal{L}(\boldsymbol{\theta})^\top \triangleq \mathbf{D}(\boldsymbol{\theta}) \quad (12.91)$$

where $\mathbf{D}(\boldsymbol{\theta})$ is called the **diffusion matrix**. Hence $\mathbb{V}[\mathbf{v}_t] = \frac{\eta}{B} \mathbf{D}(\boldsymbol{\theta}_t)$.

[LTW15] prove that the following continuous time stochastic differential equation is a first-order approximation of minibatch SGD (assuming the loss function is Lipschitz continuous):

$$d\boldsymbol{\theta}(t) = -\nabla \mathcal{L}(\boldsymbol{\theta}(t)) dt + \sqrt{\frac{\eta}{B} \mathbf{D}(\boldsymbol{\theta}_t)} d\mathbf{B}(t) \quad (12.92)$$

where $\mathbf{B}(t)$ is Brownian motion. Thus the noise from minibatching causes SGD to act like a Langevin sampler. (See [Hu+17] for more information.)

The scale factor for the noise, $\tau = \frac{\eta}{B}$, plays the role of **temperature**. Thus we see that using a smaller batch size is like using a larger temperature; the added noise ensures that SGD avoids going into narrow ravines, and instead spends most of its time in flat minima which have better generalization performance [Kes+17]. See Section 17.4.1 for more discussion of this point.

12.5.8 Applying HMC to constrained parameters

To apply HMC, we require that all the latent quantities be continuous (real-valued) and have unconstrained support, i.e., $\boldsymbol{\theta} \in \mathbb{R}^D$, so discrete latent variables need to be marginalized out (although some recent work, such as [NDL20; Zho20], relaxes this requirement).

As an example of how this can be done, consider a GMM. We can easily write the likelihood

without discrete latents as follows:

$$p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (12.93)$$

The corresponding “collapsed” model is shown in Figure 12.10(b). (Note that this is the opposite of Section 12.3.8, where we integrated out the continuous parameters in order to apply Gibbs sampling to the discrete latents.) We can apply similar techniques to other discrete latent variable models. For example, to apply HMC to HMMs, we can use the forwards algorithm (Section 9.2.2) to efficiently compute $p(\mathbf{x}_n | \boldsymbol{\theta}) = \sum_{\mathbf{z}_{1:T}} p(\mathbf{x}_n, \mathbf{z}_{n,1:T} | \boldsymbol{\theta})$.

In addition to marginalizing out any discrete latent variables, we need to ensure the remaining continuous latent variables are unconstrained. This often requires performing a change of variables using a bijector. For example, instead of sampling the discrete probability vector from the probability simplex $\boldsymbol{\pi} \in \mathbb{S}^K$, we should sample the logits $\boldsymbol{\eta} \in \mathbb{R}^K$. After sampling, we can transform back, since bijectors are invertible. (For a practical example, see [change_of_variable_hmc.ipynb](#).)

12.5.9 Speeding up HMC

Although HMC uses gradient information to explore the typical set, sometimes the geometry of the typical set can be difficult to sample from. See Section 12.5.4.3 for ways to estimate the mass matrix, which can help with such difficult cases.

Another issue is the cost of evaluating the target distribution, $\mathcal{E}(\boldsymbol{\theta}) = -\log \tilde{p}(\boldsymbol{\theta})$. For many ML applications, this has the form $\log \tilde{p}(\boldsymbol{\theta}) = \log p_0(\boldsymbol{\theta}) + \sum_{n=1}^N \log p(\boldsymbol{\theta}_n | \boldsymbol{\theta})$. This takes $O(N)$ time to compute. We can speed this up by using stochastic gradient methods; see Section 12.7 for details.

12.6 MCMC convergence

We start MCMC from an arbitrary initial state. As we explained in Section 2.6.4, the samples will be coming from the chain’s stationary distribution only when the chain has “forgotten” where it started from. The amount of time it takes to enter the stationary distribution is called the mixing time (see Section 12.6.1 for details). Samples collected before the chain has reached its stationary distribution do not come from p^* , and are usually thrown away. The initial period, whose samples will be ignored, is called the **burn-in phase**.

For example, consider a uniform distribution on the integers $\{0, 1, \dots, 20\}$. Suppose we sample from this using a symmetric random walk. In Figure 12.11, we show two runs of the algorithm. On the left, we start in state 10; on the right, we start in state 17. Even in this small problem it takes over 200 steps until the chain has “forgotten” where it started from. Proposal distributions that make larger changes can converge faster. For example, [BD92; Man] prove that it takes about 7 riffle shuffles to properly mix a deck of 52 cards (i.e., to ensure the distribution is uniform).

In Section 12.6.1 we discuss how to compute the mixing time theoretically. In practice, this can be very hard [BBM10] (this is one of the fundamental weaknesses of MCMC), so in Section 12.6.2, we discuss practical heuristics.

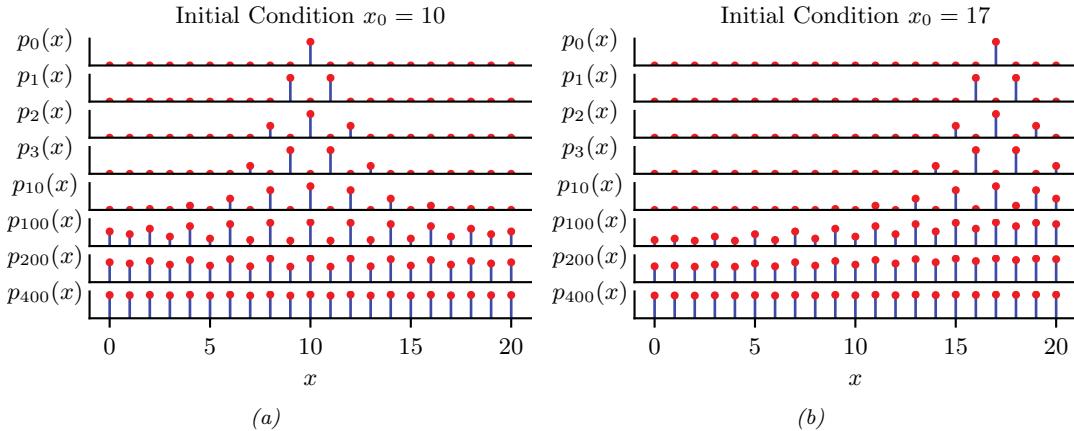


Figure 12.11: Illustration of convergence to the uniform distribution over $\{0, 1, \dots, 20\}$ using a symmetric random walk starting from (left) state 10, and (right) state 17. Adapted from Figures 29.14 and 29.15 of [Mac03]. Generated by [random_walk_integers.ipynb](#).

12.6.1 Mixing rates of Markov chains

The amount of time it takes for a Markov chain to converge to the stationary distribution, and forget its initial state, is called the **mixing time**. More formally, we say that the mixing time from state x_0 is the minimal time such that, for any constant $\epsilon > 0$, we have that

$$\tau_\epsilon(x_0) \triangleq \min\{t : \|\delta_{x_0}(x)T^t - p^*\|_1 \leq \epsilon\} \quad (12.94)$$

where $\delta_{x_0}(x)$ is a distribution with all its mass in state x_0 , T is the transition matrix of the chain (which depends on the target p^* and the proposal q), and $\delta_{x_0}(x)T^t$ is the distribution after t steps. The mixing time of the chain is defined as

$$\tau_\epsilon \triangleq \max_{x_0} \tau_\epsilon(x_0) \quad (12.95)$$

This is the maximum amount of time it takes for the chain's distribution to get ϵ close to p^* from any starting state.

The mixing time is determined by the eigengap $\gamma = \lambda_1 - \lambda_2$, which is the difference between the first and second eigenvalues of the transition matrix. For a finite state chain, one can show $\tau_\epsilon = O(\frac{1}{\gamma} \log \frac{n}{\epsilon})$, where n is the number of states.

We can also study the problem by examining the geometry of the state space. For example, consider the chain in Figure 12.12. We see that the state space consists of two “islands”, each of which is connected via a narrow “bottleneck”. (If they were completely disconnected, the chain would not be ergodic, and there would no longer be a unique stationary distribution, as discussed in Section 2.6.4.3.) We define the **conductance** ϕ of a chain as the minimum probability, over all subsets S of states, of transitioning from that set to its complement:

$$\phi \triangleq \min_{S: 0 \leq p^*(S) \leq 0.5} \frac{\sum_{x \in S, x' \in S^c} T(x \rightarrow x')}{p^*(S)}, \quad (12.96)$$

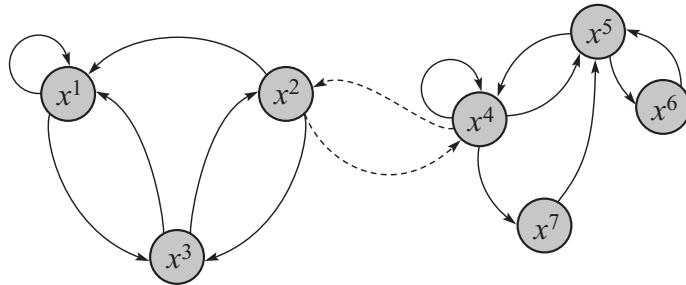


Figure 12.12: A Markov chain with low conductance. The dotted arcs represent transitions with very low probability. From Figure 12.6 of [KF09a]. Used with kind permission of Daphne Koller.

One can show that $\tau_\epsilon \leq O\left(\frac{1}{\phi^2} \log \frac{n}{\epsilon}\right)$. Hence chains with low conductance have high mixing time. For example, distributions with well-separated modes usually have high mixing time. Simple MCMC methods, such as MH and Gibbs, often do not work well in such cases, and more advanced algorithms, such as parallel tempering, are necessary (see e.g., [ED05; Kat+06; BZ20]).

12.6.2 Practical convergence diagnostics

Computing the mixing time of a chain is in general quite difficult, since the transition matrix is usually very hard to compute. Furthermore, diagnosing convergence is computationally intractable in general [BBM10]. Nevertheless, various heuristics have been proposed — see e.g., [Gey92; CC96; BR98; Veh+19]. We discuss some of the current recommended approaches below, following [Veh+19].

12.6.2.1 Trace plots

One of the simplest approaches to assessing if the method has converged is to run multiple chains (typically 3 or 4) from very different **overdispersed** starting points, and to plot the samples of some quantity of interest, such as the value of a certain component of the state vector, or some event such as the value taking on an extreme value. This is called a **trace plot**. If the chain has mixed, it should have “forgotten” where it started from, so the trace plots should converge to the same distribution, and thus overlap with each other.

To illustrate this, we will consider a very simple, but enlightening, example from [McE20, Sec 9.5]. The model is a univariate Gaussian, $y_i \sim \mathcal{N}(\alpha, \sigma)$, with just 2 observations, $y_1 = -1$ and $y_2 = +1$. We first consider a very diffuse prior, $\alpha \sim \mathcal{N}(0, 1000)$ and $\sigma \sim \text{Expon}(0.0001)$, both of which allow for very large values of α and σ . We fit the model using HMC using 3 chains and 500 samples. The result is shown in Figure 12.13. On the right, we show the trace plots for α and σ for 3 different chains. We see that they do not overlap much with each other. In addition, the numerous black vertical lines at the bottom of the plot indicate that HMC had many divergences.

The problem is caused by the overly diffuse priors, which do not get overwhelmed by the likelihood because we only have 2 datapoints. Thus the posterior is also diffuse. We can fix this by using slightly stronger priors, that keep the parameters close to more sensible values. For example, suppose we use $\alpha \sim \mathcal{N}(1, 10)$ and $\sigma \sim \text{Expon}(1)$. Now we get the results in Figure 12.14. On the right we see

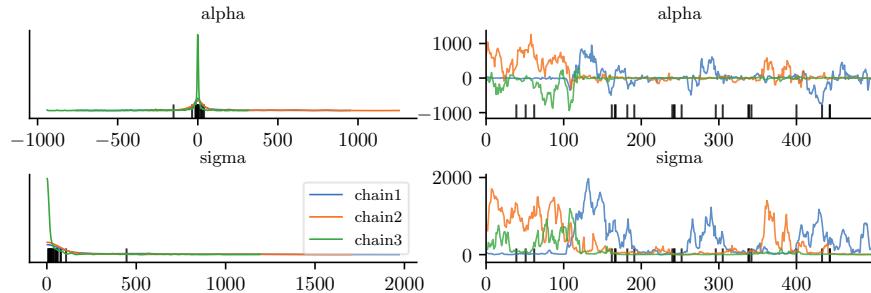


Figure 12.13: Marginals (left) and trace plot (right) for the univariate Gaussian using the diffuse prior. Black vertical lines indicate HMC divergences. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss.ipynb`.

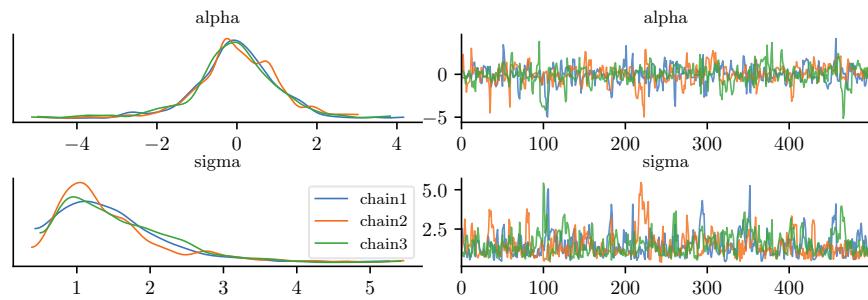


Figure 12.14: Marginals (left) and trace plot (right) for the univariate Gaussian using the sensible prior. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss.ipynb`.

that the traceplots overlap. On the left, we see that the marginal distributions from each chain have support over a reasonable interval, and have a peak at the “right” place (the MLE for α is 0, and for σ is 1). And we don’t see any divergence warnings (vertical black markers in the plot).

Since trace plots of converging chains correspond to overlapping lines, it can be hard to distinguish success from failure. An alternative plot, known as a **trace rank plot**, was recently proposed in [Veh+19]. (In [McE20], this is called a **trankplot**, a term we borrow.) The idea is to compute the rank of each sample based on all the samples from all the chains, after burnin. We then plot a histogram of the ranks for each chain separately. If the chains have converged, the distribution over ranks should be uniform, since there should be no preference for high or low scoring samples amongst the chains.

The trankplot for the model with the diffuse prior is shown in Figure 12.15. (The x-axis is from 1 to the total number of samples, which in this example is 1500, since we use 3 chains and draw 500 samples from each.) We can see that the different chains are clearly not mixing. The trankplot for the model with the sensible prior is shown in Figure 12.16; this looks much better.

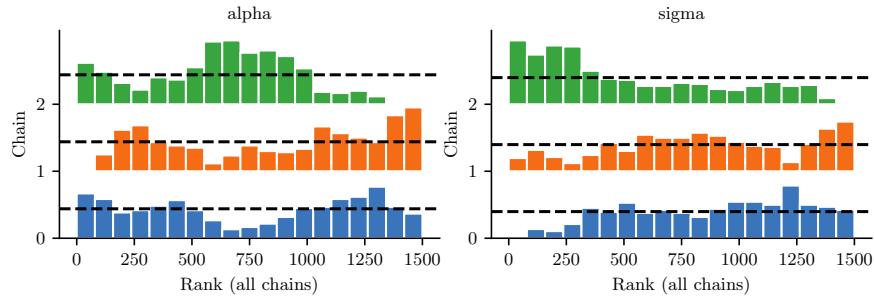


Figure 12.15: Trace rank plot for the univariate Gaussian using the diffuse prior. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss.ipynb`.

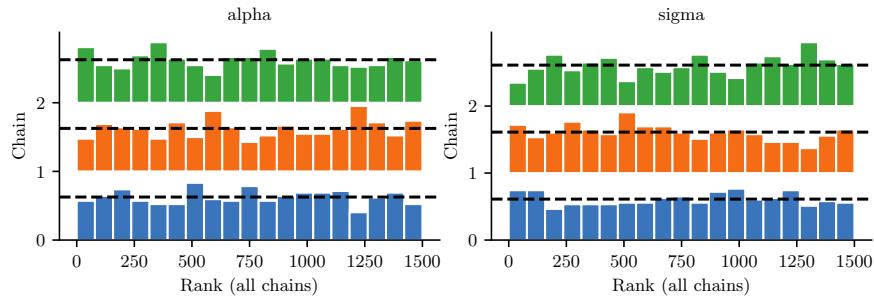


Figure 12.16: Trace rank plot for the univariate Gaussian using the sensible prior. Adapted from Figures 9.9–9.10 of [McE20]. Generated by `mcmc_traceplots_unigauss.ipynb`.

12.6.2.2 Estimated potential scale reduction (EPSR)

In this section, we discuss a way to assess convergence more quantitatively. The basic idea is this: if one or more chains has not mixed well, then the variance of all the chains combined together will be higher than the variance of the individual chains. So we will compare the variance of the quantity of interest computed between and within chains.

More precisely, suppose we have M chains, and we draw N samples from each. Let x_{nm} denote the quantity of interest derived from the n 'th sample from the m 'th chain. We compute the between-and within-sequence variances as follows:

$$B = \frac{N}{M-1} \sum_{m=1}^M (\bar{x}_{\cdot m} - \bar{x}_{\cdot \cdot})^2, \text{ where } \bar{x}_{\cdot m} = \frac{1}{N} \sum_{n=1}^N x_{nm}, \quad \bar{x}_{\cdot \cdot} = \frac{1}{M} \sum_{m=1}^M \bar{x}_{\cdot m} \quad (12.97)$$

$$W = \frac{1}{M} \sum_{m=1}^M s_m^2, \text{ where } s_m^2 = \frac{1}{N-1} \sum_{n=1}^N (x_{nm} - \bar{x}_{\cdot m})^2 \quad (12.98)$$

The formula for s_m^2 is the usual unbiased estimate for the variance from a set of N samples; W is just the average of this. The formula for B is similar, but scaled up by N since it is based on the variance of $\bar{x}_{\cdot m}$, which are averaged over N values.

Next we compute the following average variance:

$$\hat{V}^+ \triangleq \frac{N-1}{N}W + \frac{1}{N}B \quad (12.99)$$

Finally, we compute the following quantity, known as the **estimated potential scale reduction** or **R-hat**:

$$\hat{R} \triangleq \sqrt{\frac{\hat{V}^+}{W}} \quad (12.100)$$

In [Veh+19], they recommend checking if $\hat{R} < 1.01$ before declaring convergence.

For example, consider the \hat{R} values for various samplers for our univariate GMM example. In particular, consider the 3 MH samplers in Figure 12.1, and the Gibbs sampler in Figure 12.4. The \hat{R} values are 1.493, 1.039, 1.005, and 1.007. So this diagnostic has correctly identified that the first two samplers are unreliable, which is evident from the figure.

In practice, it is recommended to use a slightly different quantity, known as **split- \hat{R}** . This can be computed by splitting each chain into the first and second halves, thus doubling the number of chains M (but halving the number of samples N from each), before computing \hat{R} . This can detect non-stationarity within a single chain.

12.6.3 Effective sample size

Although MCMC lets us draw samples from a target distribution (assuming it has converged), the samples are not independent, so we may need to draw a lot of them to get a reliable estimate. In this section, we discuss how to compute the **effective sample size** or **ESS** from a set of (possibly correlated) samples.

To start, suppose we draw N *independent* samples from the target distribution, and let $\hat{x} = \frac{1}{N} \sum_{n=1}^N x_n$ be our empirical estimate of the mean of the quantity of interest. The variance of this estimate is given by

$$\mathbb{V}[\hat{x}] = \frac{1}{N^2} \mathbb{V}\left[\sum_{n=1}^N x_n\right] = \frac{1}{N^2} \sum_{n=1}^N \mathbb{V}[x_n] = \frac{1}{N} \sigma^2 \quad (12.101)$$

where $\sigma^2 = \mathbb{V}[X]$. If the samples are correlated, the variance of the estimate will be higher, as we show below.

Recall that for N (not necessarily independent) random variables we have

$$\mathbb{V}\left[\sum_{n=1}^N x_n\right] = \sum_{i=1}^N \sum_{j=1}^N \text{Cov}[x_i, x_j] = \sum_{i=1}^N \mathbb{V}[x_i] + 2 \sum_{1 \leq i < j \leq N} \text{Cov}[x_i, x_j] \quad (12.102)$$

Let $\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$ be our estimate based on these correlated samples. The variance of this estimate is given by

$$\mathbb{V}[\bar{x}] = \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N \text{Cov}[x_i, x_j] \quad (12.103)$$

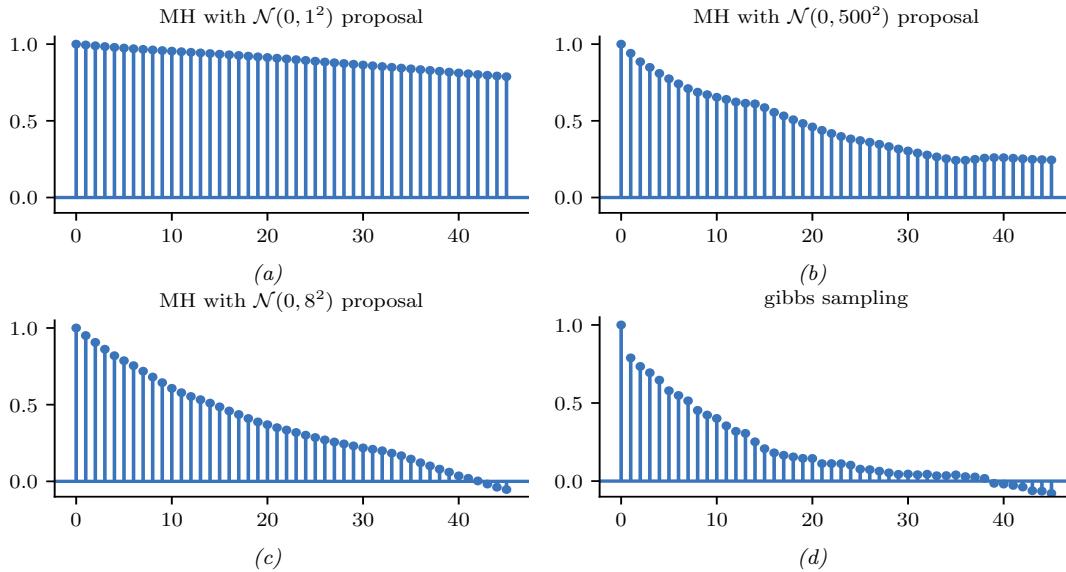


Figure 12.17: Autocorrelation functions for various MCMC samplers for the mixture of two 1d Gaussians. (a-c) These are the MH samplers in Figure 12.1. (d) This is the Gibbs sampler in Figure 12.4. Generated by `mcmc_gmm_demo.ipynb`.

We now rewrite this in a more convenient form. First recall that the correlation of x_i and x_j is given by

$$\text{corr}[x_i, x_j] = \frac{\text{Cov}[x_i, x_j]}{\sqrt{\mathbb{V}[x_i] \mathbb{V}[x_j]}} \quad (12.104)$$

Since we assume we are drawing samples from the target distribution, we have $\mathbb{V}[x_i] = \sigma^2$, where σ^2 is the true variance. Hence

$$\mathbb{V}[\bar{x}] = \frac{\sigma^2}{N^2} \sum_{i=1}^N \sum_{j=1}^N \text{corr}[x_i, x_j] \quad (12.105)$$

For a fixed i , we can think of $\text{corr}[x_i, x_j]$ as a function of j . This will usually decay as j gets further from i . As $N \rightarrow \infty$ we can approximate the sum of correlations by

$$\sum_{j=1}^N \text{corr}[x_i, x_j] \rightarrow \sum_{\ell=-\infty}^{\infty} \text{corr}[x_i, x_{i+\ell}] = 1 + 2 \sum_{\ell=1}^{\infty} \text{corr}[x_i, x_{i+\ell}] \quad (12.106)$$

since $\text{corr}[x_i, x_i] = 1$ and $\text{corr}[x_i, x_{i-\ell}] = \text{corr}[x_i, x_{i+\ell}]$ for lag $\ell > 0$. Since we assume the samples are coming from a stationary distribution, the index i does not matter. Thus we can define the **autocorrelation time** as

$$\rho = 1 + 2 \sum_{\ell=1}^{\infty} \rho(\ell) \quad (12.107)$$

where $\rho(\ell)$ is the **autocorrelation function** (ACF), defined as

$$\rho(\ell) \triangleq \text{corr} [x_0, x_\ell] \quad (12.108)$$

The ACF can be approximated efficiently by convolving the signal \boldsymbol{x} with itself. In Figure 12.17, we plot the ACF for our four samplers for the GMM. We see that the ACF of the Gibbs sampler (bottom right) dies off to 0 much more rapidly than the MH samplers. Intuitively this indicates that each Gibbs sample is “worth” more than each MH sample. We quantify this below.

From Equation (12.105), we can compute the variance of our estimate in terms of the ACF as follows: $\mathbb{V}[\bar{x}] = \frac{\sigma^2}{N^2} \sum_{i=1}^N \rho = \frac{\sigma^2}{N} \rho$. By contrast, the variance of the estimate from independent samples is $\mathbb{V}[\hat{x}] = \frac{\sigma^2}{N}$. So we see that the variance is a factor ρ larger when there is correlation. We therefore define the **effective sample size** of our set of samples to be

$$N_{\text{eff}} \triangleq \frac{N}{\rho} = \frac{N}{1 + 2 \sum_{\ell=1}^{\infty} \rho(\ell)} \quad (12.109)$$

In practice, we truncate the sum at lag L , which is the last integer at which $\rho(L)$ is positive. Also, if we run M chains, the numerator should be NM , so we get the following estimate:

$$\hat{N}_{\text{eff}} = \frac{NM}{1 + 2 \sum_{\ell=1}^L \hat{\rho}(\ell)} \quad (12.110)$$

In [Veh+19], they propose various extensions of the above estimator, such as using rank statistics, to make the estimate more robust.

12.6.4 Improving speed of convergence

There are many possible things you could try if the \hat{R} value is too large, and/or the effective sample size is too low. Here is a brief list:

- Try using a non-centered parameterization (see Section 12.6.5).
- Try sampling variables in groups or blocks (see Section 12.3.7).
- Try using Rao-Blackwellization, i.e., analytically integrating out some of the variables (see Section 12.3.8).
- Try adding auxiliary variables (see Section 12.4).
- Try using adaptive proposal distributions (see Section 12.2.3.5).

More details can be found in [Rob+18].

12.6.5 Non-centered parameterizations and Neal’s funnel

A common problem that arises when applying sampling to hierarchical Bayesian models is when a set of parameters at one level of the model have a tight dependence on parameters at the level above. We show some practical examples of this in the hierarchical Gaussian 8-schools example in

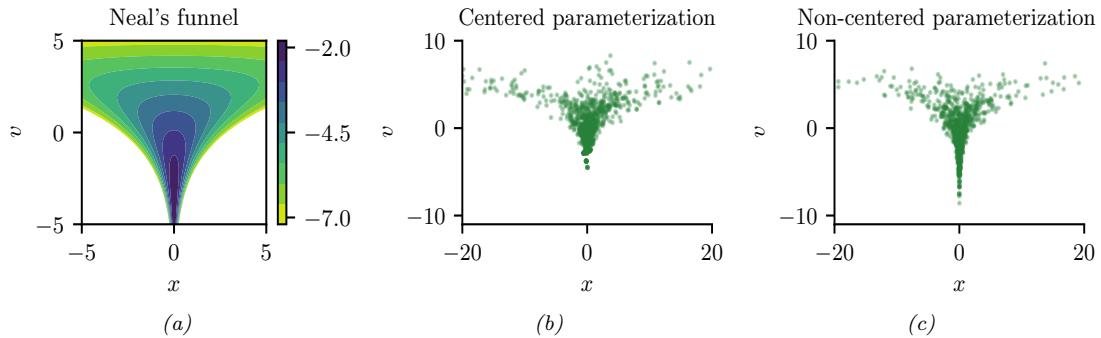


Figure 12.18: Neal’s funnel. (a) Joint density. (b) HMC samples from centered representation. (c) HMC samples from non-centered representation. Generated by [neals_funnel.ipynb](#).

Section 3.6.2.2 and the hierarchical radon regression example in Section 15.5.2.2. Here, we focus on the following simple toy model that captures the essence of the problem:

$$\nu \sim \mathcal{N}(0, 3) \tag{12.111}$$

$$x \sim \mathcal{N}(0, \exp(\nu)) \tag{12.112}$$

The corresponding joint density $p(x, \nu)$ is shown in Figure 12.18a. This is known **Neal’s funnel**, named after [Nea03]. It is hard for a sampler to “descend” in the narrow “neck” of the distribution, corresponding to areas where the variance ν is small [BG13].

Fortunately, we can represent this model in an equivalent way that makes it easier to sample from, providing we use a **non-centered parameterization** [PR03]. This has the form

$$\nu \sim \mathcal{N}(0, 3) \tag{12.113}$$

$$z \sim \mathcal{N}(0, 1) \tag{12.114}$$

$$x = z \exp(\nu) \tag{12.115}$$

This is easier to sample from, since $p(z, \nu)$ is a product of 2 independent Gaussians, and we can derive x deterministically from these Gaussian samples. The advantage of this reparameterization is shown in Figure 12.18. A method to automatically derive such reparameterizations is discussed in [GMH20].

12.7 Stochastic gradient MCMC

Consider an unnormalized target distribution of the following form:

$$\pi(\boldsymbol{\theta}) \propto p(\boldsymbol{\theta}, \mathcal{D}) = p_0(\boldsymbol{\theta}) \prod_{n=1}^N p(\mathbf{x}_n | \boldsymbol{\theta}) \tag{12.116}$$

where $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$. Alternatively we can define the target distribution in terms of an energy function (negative log joint) as follows:

$$p(\boldsymbol{\theta}, \mathcal{D}) \propto \exp(-\mathcal{E}(\boldsymbol{\theta})) \tag{12.117}$$

The energy function can be decomposed over data samples:

$$\mathcal{E}(\boldsymbol{\theta}) = \sum_{n=1}^N \mathcal{E}_n(\boldsymbol{\theta}) \quad (12.118)$$

$$\mathcal{E}_n(\boldsymbol{\theta}) = -\log p(\mathbf{x}_n | \boldsymbol{\theta}) - \frac{1}{N} \log p_0(\boldsymbol{\theta}) \quad (12.119)$$

Evaluating the full energy (e.g., to compute an acceptance probability in the Metropolis-Hastings algorithm, or to compute the gradient in HMC) takes $O(N)$ time, which does not scale to large data. In this section, we discuss some solutions to this problem.

12.7.1 Stochastic gradient Langevin dynamics (SGLD)

Recall from Equation (12.83) that the **Langevin diffusion** SDE has the following form

$$d\boldsymbol{\theta}_t = -\nabla \mathcal{E}(\boldsymbol{\theta}_t) dt + \sqrt{2} d\mathbf{W}_t \quad (12.120)$$

where $d\mathbf{W}_t$ is a Wiener noise (also called Brownian noise) process. In discrete time, we can use the following Euler approximation:

$$\boldsymbol{\theta}_{t+1} \approx \boldsymbol{\theta}_t - \eta_t \nabla \mathcal{E}(\boldsymbol{\theta}_t) + \sqrt{2\eta_t} \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (12.121)$$

Computing the gradient $\mathbf{g}(\boldsymbol{\theta}_t) = \nabla \mathcal{E}(\boldsymbol{\theta}_t)$ at each step takes $O(N)$ time. We can compute an unbiased minibatch approximation to the gradient term in $O(B)$ time using

$$\hat{\mathbf{g}}(\boldsymbol{\theta}_t) = \frac{N}{B} \sum_{n \in \mathcal{B}_t} \nabla \mathcal{E}_n(\boldsymbol{\theta}_t) = -\frac{N}{B} \left(\sum_{n \in \mathcal{B}_t} \nabla \log p(\mathbf{x}_n | \boldsymbol{\theta}_t) + \frac{B}{N} \nabla \log p_0(\boldsymbol{\theta}_t) \right) \quad (12.122)$$

where \mathcal{B}_t is the minibatch at step t . This gives rise to the following approximate update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \hat{\mathbf{g}}(\boldsymbol{\theta}_t) + \sqrt{2\eta_t} \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (12.123)$$

This is called **stochastic gradient Langevin dynamics** or **SGLD** [Wei11]. The resulting update step is identical to SGD, except for the addition of a Gaussian noise term. (See [Neg+21] for some recent analysis of this method; they also suggest setting $\eta_t \propto N^{-2/3}$.)

12.7.2 Preconditioning

As in SGD, we can get better results (especially for models such as neural networks) if we use preconditioning to scale the gradient updates. In [PT13], they use the Fisher information matrix (FIM) as the preconditioner; this method is known as **stochastic gradient Riemannian Langevin dynamics** or **SGRLD**.

Unfortunately, computing the FIM is often hard. In [Li+16], they propose to use the same kind of diagonal approximation as used by RMSprop; this is called **preconditioned SGLD**. An alternative is to use an Adam-like preconditioner, as proposed in [KSL21]. This is called **SGLD-Adam**. For more details, see [CSN21].

12.7.3 Reducing the variance of the gradient estimate

The variance of the noise introduced by minibatching can be quite large, which can hurt the performance of methods such as SGLD [BDM18]. In [Bak+17], they propose to reduce the variance of this estimate by using a **control variate** estimator; this method is therefore called **SGLD-CV**. Specifically they use the following gradient approximation:

$$\hat{\nabla}_{cv}\mathcal{E}(\boldsymbol{\theta}_t) = \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}})) \quad (12.124)$$

Here $\hat{\boldsymbol{\theta}}$ is any fixed value, but it is often taken to be an approximate MAP estimate (e.g., based on one epoch of SGD). The reason Equation (12.124) is valid is because the terms we add and subtract are equal in expectation, and hence we get an unbiased estimate:

$$\mathbb{E} [\hat{\nabla}_{cv}\mathcal{E}(\boldsymbol{\theta}_t)] = \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \mathbb{E} \left[\frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}})) \right] \quad (12.125)$$

$$= \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) + \nabla\mathcal{E}(\boldsymbol{\theta}_t) - \nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) = \nabla\mathcal{E}(\boldsymbol{\theta}_t) \quad (12.126)$$

Note that the first term, $\nabla\mathcal{E}(\hat{\boldsymbol{\theta}}) = \sum_{n=1}^N \nabla\mathcal{E}_n(\hat{\boldsymbol{\theta}})$, requires a single pass over the entire dataset, but only has to be computed once (e.g., while estimating $\hat{\boldsymbol{\theta}}$).

One disadvantage of SGLD-CV is that the reference point $\hat{\boldsymbol{\theta}}$ has to be precomputed, and is then fixed. An alternative is to update the reference point online, by performing periodic full batch estimates. This is called **SVRG-LD** [Dub+16; Cha+18], where SVRG stands for stochastic variance reduced gradient, and LD stands for Langevin dynamics. If we use $\tilde{\boldsymbol{\theta}}_t$ to denote the most recent snapshot (reference point), the corresponding gradient estimate is given by

$$\hat{\nabla}_{svrg}\mathcal{E}(\boldsymbol{\theta}_t) = \nabla\mathcal{E}(\tilde{\boldsymbol{\theta}}_t) + \frac{N}{B} \sum_{n \in \mathcal{S}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\tilde{\boldsymbol{\theta}}_t)) \quad (12.127)$$

We recompute the snapshot every τ steps (known as the epoch length). See Algorithm 12.5 for the pseudo-code.

Algorithm 12.5: SVRG Langevin descent

```

1 Initialize  $\boldsymbol{\theta}_0$ 
2 for  $t = 1 : T$  do
3   if  $t \bmod \tau = 0$  then
4      $\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}_t$ 
5      $\tilde{\mathbf{g}} = \sum_{n=1}^N \mathcal{E}_n(\tilde{\boldsymbol{\theta}})$ 
6     Sample minibatch  $\mathcal{B}_t \in \{1, \dots, N\}$ 
7      $\mathbf{g}_t = \tilde{\mathbf{g}} + \frac{N}{B} \sum_{n \in \mathcal{B}_t} (\nabla\mathcal{E}_n(\boldsymbol{\theta}_t) - \nabla\mathcal{E}_n(\tilde{\boldsymbol{\theta}}))$ 
8      $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{g}_t + \sqrt{2\eta_t} \mathcal{N}(\mathbf{0}, \mathbf{I})$ 

```

The disadvantage of SVRG is that it needs to perform a full pass over the data every τ steps. An alternative approach, called **SAGA-LD** [Dub+16; Cha+18] (which stands for stochastic averaged gradient acceleration), avoids this by storing all N gradient vectors, and then doing incremental updates. Unfortunately the memory requirements of this algorithm usually make it impractical.

12.7.4 SG-HMC

We discussed Hamiltonian Monte Carlo (HMC) in Section 12.5, which uses auxiliary momentum variables to improve performance over Langevin MC. In this section, we discuss a way to speed it up by approximating the gradients using minibatches. This is called **SG-HMC** [CFG14; ZG21], where SG stands for “stochastic gradient”.

Recall that the leapfrog updates have the following form:

$$\mathbf{v}_{t+1/2} = \mathbf{v}_t - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_t) \quad (12.128)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \mathbf{v}_{t+1/2} = \boldsymbol{\theta}_t + \eta \mathbf{v}_t - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_t) \quad (12.129)$$

$$\mathbf{v}_{t+1} = \mathbf{v}_{t+1/2} - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_{t+1}) = \mathbf{v}_t - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_t) - \frac{\eta}{2} \nabla \mathcal{E}(\boldsymbol{\theta}_{t+1}) \quad (12.130)$$

We can replace the full batch gradient with a stochastic approximation, to get

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \mathbf{v}_t - \frac{\eta^2}{2} \mathbf{g}(\boldsymbol{\theta}_t, \boldsymbol{\xi}_t) \quad (12.131)$$

$$\mathbf{v}_{t+1} = \mathbf{v}_t - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_t, \boldsymbol{\xi}_t) - \frac{\eta}{2} \mathbf{g}(\boldsymbol{\theta}_{t+1}, \boldsymbol{\xi}_{t+1/2}) \quad (12.132)$$

where $\boldsymbol{\xi}_t$ and $\boldsymbol{\xi}_{t+1/2}$ are independent sources of randomness (e.g., batch indices). In [ZG21], they show that this algorithm (even without the MH rejection step) provides a good approximation to the posterior (in the sense of having small Wasserstein-2 distance) for the case where the energy function is strongly convex. Furthermore, performance can be considerably improved if we use the variance reduction methods discussed in Section 12.7.3.

12.7.5 Underdamped Langevin dynamics

The **underdamped Langevin dynamics (ULD)** has the form of the following SDE [CDC15; LMS16; Che+18a; Che+18d]:

$$\begin{aligned} d\boldsymbol{\theta}_t &= \mathbf{v}_t dt \\ d\mathbf{v}_t &= -\mathbf{g}(\boldsymbol{\theta}_t) dt - \gamma \mathbf{v}_t dt + \sqrt{2\gamma} d\mathbf{W}_t \end{aligned} \quad (12.133)$$

where $\mathbf{g}(\boldsymbol{\theta}_t) = \nabla \mathcal{E}(\boldsymbol{\theta}_t)$ is the gradient or **force** acting on the particle, $\gamma > 0$ is the **friction** parameter, and $d\mathbf{W}_t$ is Wiener noise.

Equation (12.133) is like the Langevin dynamics of Equation (12.83) but with an added momentum term \mathbf{v}_t . We can solve the dynamics using various integration methods. It can be shown (see e.g., [LMS16]) that these methods are accurate to second order, whereas solving standard (overdamped) Langevin is only accurate to first order, and thus will require more sampling steps to achieve a given accuracy.

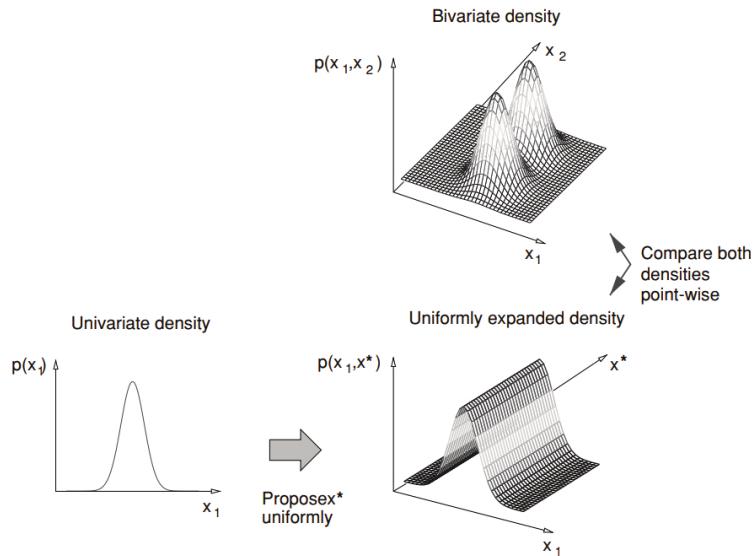


Figure 12.19: To compare a 1d model against a 2d model, we first have to map the 1d model to 2d space so the two have a common measure. Note that we assume the ridge has finite support, so it is integrable. From Figure 17 of [And+03]. Used with kind permission of Nando de Freitas.

12.8 Reversible jump (transdimensional) MCMC

Suppose we have a set of models with different numbers of parameters, e.g., mixture models in which the number of mixture components is unknown. Let the model be denoted by m , and let its unknowns (e.g., parameters) be denoted by $\boldsymbol{x}_m \in \mathcal{X}_m$ (e.g., $\mathcal{X}_m = \mathbb{R}^{n_m}$, where n_m is the dimensionality of model m). Sampling in spaces of differing dimensionality is called **trans-dimensional MCMC**. We could sample the model indicator $m \in \{1, \dots, M\}$ and sample all the parameters from the product space $\prod_{m=1}^M \mathcal{X}_m$, but this is very inefficient, and only works if M is finite. It is more parsimonious to sample in the union space $\mathcal{X} = \cup_{m=1}^M \{m\} \times \mathcal{X}_m$, where we only worry about parameters for the currently active model.

The difficulty with this approach arises when we move between models of different dimensionality. The trouble is that when we compute the MH acceptance ratio, we are comparing densities defined on spaces of different dimensionality, which is not well defined. For example, comparing densities on two points of a sphere makes sense, but comparing a density on a sphere to a density on a circle does not, as there is a dimensional mismatch in the two concepts. The solution, proposed by [Gre95] and known as **reversible jump MCMC** or **RJMCMC**, is to augment the low dimensional space with extra random variables so that the two spaces have a common measure. This is illustrated in Figure 12.19.

We give a sketch of the algorithm below. For more details, see e.g., [Gre03; HG12].

12.8.1 Basic idea

To explain the method in more detail, we follow the presentation of [And+03]. To ensure a common measure, we need to define a way to extend each pair of subspaces \mathcal{X}_m and \mathcal{X}_n to $\mathcal{X}_{m,n} = \mathcal{X}_m \times \mathcal{U}_{m,n}$ and $\mathcal{X}_{n,m} = \mathcal{X}_n \times \mathcal{U}_{n,m}$. We also need to define a deterministic, differentiable and invertible mapping

$$(\mathbf{x}_m, \mathbf{u}_{m,n}) = f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m}) = (f_{n \rightarrow m}^x(\mathbf{x}_n, \mathbf{u}_{n,m}), f_{n \rightarrow m}^u(\mathbf{x}_n, \mathbf{u}_{n,m})) \quad (12.134)$$

Invertibility means that

$$f_{m \rightarrow n}(f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m})) = (\mathbf{x}_n, \mathbf{u}_{n,m}) \quad (12.135)$$

Finally, we need to define proposals $q_{n \rightarrow m}(\mathbf{u}_{n,m}|n, \mathbf{x}_n)$ and $q_{m \rightarrow n}(\mathbf{u}_{m,n}|m, \mathbf{x}_m)$.

Suppose we are in state (n, \mathbf{x}_n) . We move to (m, \mathbf{x}_m) by generating $\mathbf{u}_{n,m} \sim q_{n \rightarrow m}(\cdot|n, \mathbf{x}_n)$, and then computing $(\mathbf{x}_m, \mathbf{u}_{m,n}) = f_{n \rightarrow m}(\mathbf{x}_n, \mathbf{u}_{n,m})$. We then accept the move with probability

$$A_{n \rightarrow m} = \min \left\{ 1, \frac{p(m, \mathbf{x}_m^*)}{p(n, \mathbf{x}_n)} \times \frac{q(n|m)}{q(m|n)} \times \frac{q_{m \rightarrow n}(\mathbf{u}_{m,n}|m, \mathbf{x}_m^*)}{q_{n \rightarrow m}(\mathbf{u}_{n,m}|n, \mathbf{x}_n)} \times |\det \mathbf{J}_{f_{m \rightarrow n}}| \right\} \quad (12.136)$$

where $\mathbf{x}_m^* = f_{n \rightarrow m}^x(\mathbf{x}_n, \mathbf{u}_{n,m})$, $\mathbf{J}_{f_{m \rightarrow n}}$ is the Jacobian of the transformation

$$J_{f_{m \rightarrow n}} = \frac{\partial f_{n \rightarrow m}(\mathbf{x}_m, \mathbf{u}_{m,n})}{\partial (\mathbf{x}_m, \mathbf{u}_{m,n})} \quad (12.137)$$

and $|\det \mathbf{J}|$ is the absolute value of the determinant of the Jacobian.

12.8.2 Example

Let us consider an example from [AFD01]. They consider an RBF network for nonlinear regression of the form

$$f(\mathbf{x}) = \sum_{j=1}^k a_j \mathcal{K}(\|\mathbf{x} - \boldsymbol{\mu}_j\|) + \boldsymbol{\beta}^\top \mathbf{x} + \beta_0 + \epsilon \quad (12.138)$$

where $\mathcal{K}()$ is some kernel function (e.g., a Gaussian), k is the number of such basis functions, and ϵ is a Gaussian noise term. If $k = 0$, the model corresponds to linear regression.

They fit this model to the data in Figure 12.20(a). The predictions on the test set are shown in Figure 12.20(b). Estimates of $p(k|\mathcal{D})$, the (distribution over the) number of basis functions, are shown in Figure 12.20(c) as a function of the iteration number; the posterior at the final iteration is shown in Figure 12.20(d). There is clearly the most posterior support for $k = 2$, which makes sense given the two ‘‘bumps’’ in the data.

To generate these results, they consider several kinds of proposal. One of them is to split a current basis function μ into two new ones using

$$\mu_1 = \mu - u_{n,n+1}\alpha, \quad \mu_2 = \mu + u_{n,n+1}\alpha \quad (12.139)$$

where α is a parameter of the proposal, and $u_{n,m}$ is sampled from some distribution (e.g., uniform). To ensure reversibility, they define a corresponding merge move

$$\mu = \frac{\mu_1 + \mu_2}{2} \quad (12.140)$$

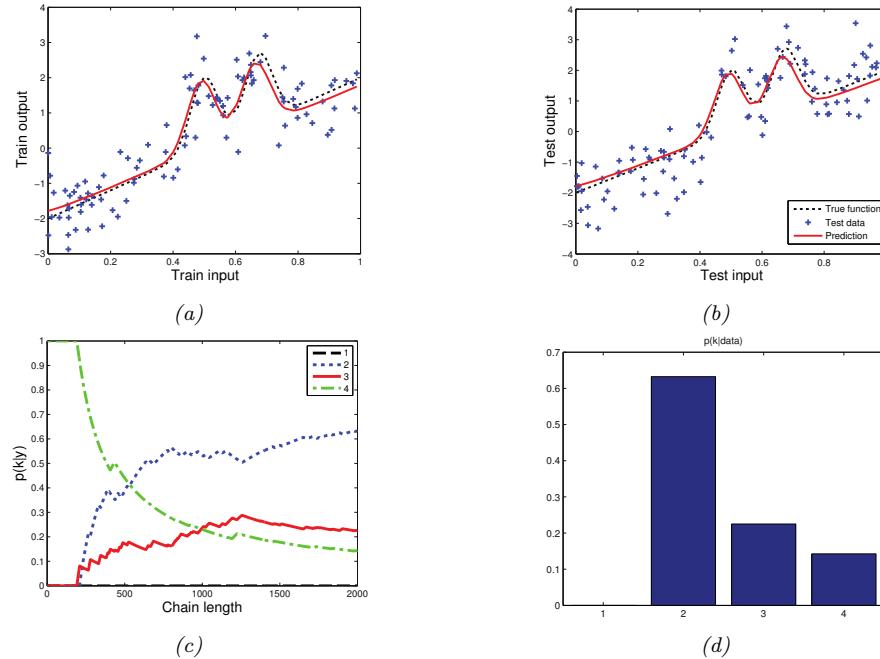


Figure 12.20: Fitting an RBF network to some 1d data using RJMCMC. (a) Prediction on train set. (b) Prediction on test set. (c) Plot of $p(k|D)$ vs iteration. (d) Final posterior $p(k|D)$. Adapted from Figure 4 of [AFD01]. Generated by `rjmcme_rbf`, written by Nando de Freitas.

where μ_1 is chosen at random, and μ_2 is its nearest neighbor. To ensure these moves are reversible, we require $\|\mu_1 - \mu_2\| < 2\beta$.

The acceptance ratio for the split move is given by

$$A_{\text{split}} = \min \left\{ 1, \frac{p(k+1, \mu_{k+1})}{p(k, \mu_k)} \times \frac{1/(k+1)}{1/k} \times \frac{1}{p(u_{n,m})} \times |\det \mathbf{J}_{\text{split}}| \right\} \quad (12.141)$$

where $1/k$ is the probability of choosing one of the k bases uniformly at random. The Jacobian is

$$\mathbf{J}_{\text{split}} = \frac{\partial(\mu_1, \mu_2)}{\partial(\mu, u_{n,m})} = \det \begin{pmatrix} 1 & 1 \\ -\beta & \beta \end{pmatrix} \quad (12.142)$$

so $|\det \mathbf{J}_{\text{split}}| = 2\beta$. The acceptance ratio for the merge move is given by

$$A_{\text{merge}} = \min \left\{ 1, \frac{p(k-1, \mu_{k-1})}{p(k, \mu_k)} \times \frac{1/(k-1)}{1/k} \times |\det \mathbf{J}_{\text{merge}}| \right\} \quad (12.143)$$

where $|\det \mathbf{J}_{\text{merge}}| = 1/(2\beta)$.

The overall pseudo-code for the algorithm, assuming the current model has index k , is given in Algorithm 12.6. Here b_k is the probability of a birth move, d_k is the probability of a death move, s_k

Algorithm 12.6: Generic reversible jump MCMC (single step)

- 1 Sample $u \sim U(0, 1)$
 - 2 If $u \leq b_k$
 - 3 then birth move
 - 4 else if $u \leq (b_k + d_k)$ then death move
 - 5 else if $u \leq (b_k + d_k + s_k)$ then split move
 - 6 else if $u \leq (b_k + d_k + s_k + m_k)$ then merge move
 - 7 else update parameters
-

is the probability of a split move, and m_k is the probability of a merge move. If we don't make a dimension-changing move, we just update the parameters of the current model using random walk MH.

12.8.3 Discussion

RJMCMC algorithms can be quite tricky to implement. If, however, the continuous parameters can be integrated out (resulting in a method called collapsed RJMCMC), much of the difficulty goes away, since we are just left with a discrete state space, where there is no need to worry about change of measure. For example, if we fix the centers μ_j in Equation (12.138) (e.g., using samples from the data, or using K-means clustering), we are left with a linear model, where we can integrate out the parameters. All that is left to do is sample which of these fixed basis functions to include in the model, which is a discrete variable selection problem. See e.g., [Den+02] for details.

In Chapter 31, we discuss Bayesian nonparametric models, which allow for an infinite number of different models. Surprisingly, such models are often easier to deal with computationally (as well as more realistic, statistically) than working with a finite set of different models.

12.9 Annealing methods

Many distributions are multimodal and hence hard to sample from. However, by analogy to the way metals are heated up and then cooled down in order to make the molecules align, we can imagine using a computational temperature parameter to "smooth out" a distribution, gradually cooling it to recover the original "bumpy" distribution. We first explain this idea in more detail in the context of an algorithm for MAP estimation. We then discuss extensions to the sampling case.

12.9.1 Simulated annealing

In this section, we discuss the **simulated annealing** algorithm [KJV83; LA87], which is a variant of the Metropolis-Hastings algorithm which is designed to find the global optimum of blackbox function. (Other approaches to blackbox optimization are discussed in Section 6.7.)

Annealing is a physical process of heating a solid until thermal stresses are released, then cooling it very slowly until the crystals are perfectly arranged, achieving a minimum energy state. Depending on how fast or slow the temperature is cooled, the results will have better or worse quality. We can apply this approach to probability distributions, to control the number of modes (low energy states)

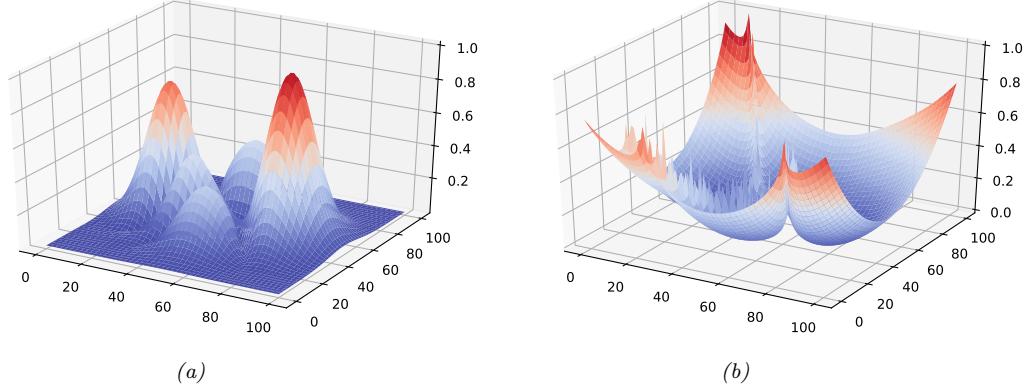


Figure 12.21: (a) A peaky distribution. (b) Corresponding energy function. Generated by [simulated_annealing_2d_demo.ipynb](#).

that they have, by defining

$$p_T(\mathbf{x}) = \exp(-\mathcal{E}(\mathbf{x})/T) \quad (12.144)$$

where T is the temperature, which is reduced over time. As an example, consider the **peaks** function:

$$p(x, y) \propto |3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\frac{x}{5} - x^3 - y^5) e^{-x^2-y^2} - \frac{1}{3} e^{-(x+1)^2-y^2}| \quad (12.145)$$

This is plotted in Figure 12.21a. The corresponding energy is in Figure 12.21b. We plot annealed versions of this distribution in Figure 12.22. At high temperatures, $T \gg 1$, the surface is approximately flat, and hence it is easy to move around (i.e., to avoid local optima). As the temperature cools, the largest peaks become larger, and the smallest peaks disappear. By cooling slowly enough, it is possible to “track” the largest peak, and thus find the global optimum (minimum energy state). This is an example of a **continuation method**.

In more detail, at each step, we sample a new state according to some proposal distribution $\mathbf{x}' \sim q(\cdot | \mathbf{x}_t)$. For real-valued parameters, this is often simply a random walk proposal centered on the current iterate, $\mathbf{x}' = \mathbf{x}_t + \boldsymbol{\epsilon}_{t+1}$, where $\boldsymbol{\epsilon}_{t+1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. (The matrix $\boldsymbol{\Sigma}$ is often diagonal, and may be updated over time using the method in [Cor+87].) Having proposed a new state, we compute the acceptance probability

$$\alpha_{t+1} = \exp(-(E(\mathbf{x}') - E(\mathbf{x}_t))/T_t) \quad (12.146)$$

where T_t is the temperature of the system. We then accept the new state (i.e., set $\mathbf{x}_{t+1} = \mathbf{x}'$) with probability $\min(1, \alpha_{t+1})$, otherwise we stay in the current state (i.e., set $\mathbf{x}_{t+1} = \mathbf{x}_t$). This means that if the new state has lower energy (is more probable), we will definitely accept it, but if it has higher energy (is less probable), we might still accept, depending on the current temperature. Thus the algorithm allows “downhill” moves in probability space (uphill in energy space), but less frequently as the temperature drops.

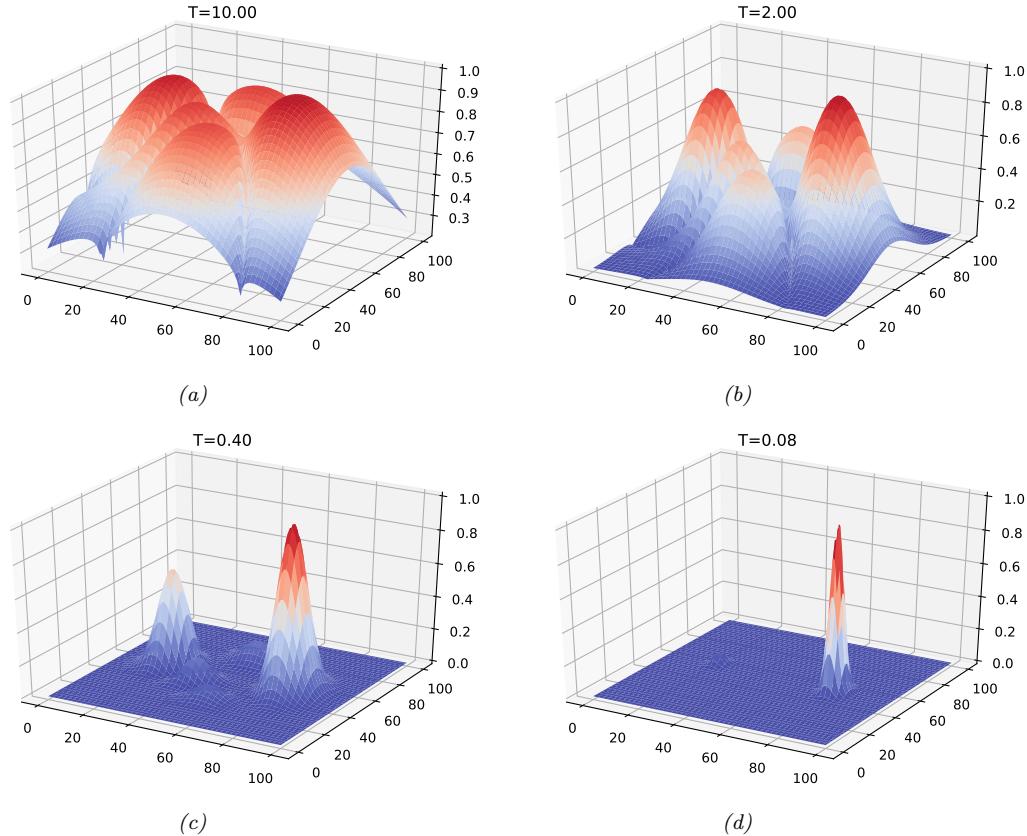


Figure 12.22: Annealed version of the distribution in Figure 12.21a at different temperatures. Generated by [simulated_annealing_2d_demo.ipynb](#).

The rate at which the temperature changes over time is called the **cooling schedule**. It has been shown [Haj88] that if one cools according to a logarithmic schedule, $T_t \propto 1/\log(t+1)$, then the method is guaranteed to find the global optimum under certain assumptions. However, this schedule is often too slow. In practice it is common to use an **exponential cooling schedule** of the form $T_{t+1} = \gamma T_t$, where $\gamma \in (0, 1]$ is the cooling rate. Cooling too quickly means one can get stuck in a local maximum, but cooling too slowly just wastes time. The best cooling schedule is difficult to determine; this is one of the main drawbacks of simulated annealing.

In Figure 12.23a, we show a cooling schedule using $\gamma = 0.9$. If we combine this with a Gaussian random walk proposal with $\sigma = 10$ to the peaky distribution in Figure 12.21a, we get the results shown in Figure 12.23 and Figure 12.23b. We see that the algorithm concentrates its samples near the global optimum (the peak on the middle right).

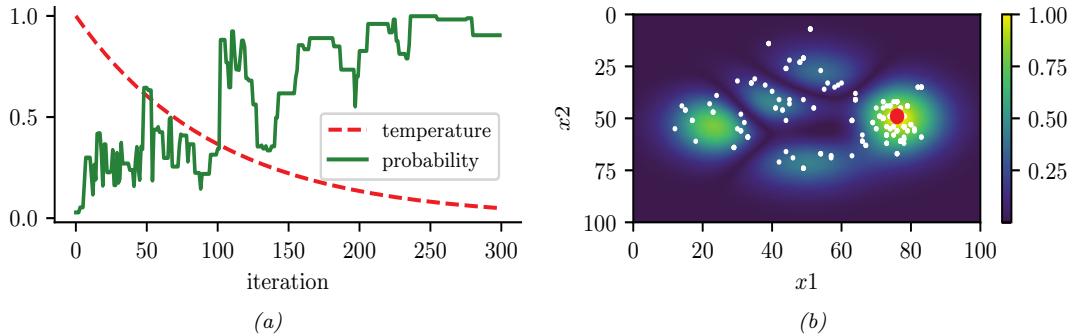


Figure 12.23: Simulated annealing applied to the distribution in Figure 12.21a. (a) Temperature vs iteration and probability of each visited point vs iteration. (b) Visited samples, superimposed on the target distribution. The big red dot is the highest probability point found. Generated by [simulated_annealing_2d_demo.ipynb](#).

12.9.2 Parallel tempering

Another way to combine MCMC and annealing is to run multiple chains in parallel at different temperatures, and allow one chain to sample from another chain at a neighboring temperature. In this way, the high temperature chain can make long distance moves through the state space, and have this influence lower temperature chains. This is known as **parallel tempering**. See e.g., [ED05; Kat+06] for details.

13 Sequential Monte Carlo

13.1 Introduction

In this chapter, we discuss **sequential Monte Carlo** or **SMC** algorithms, which can be used to sample from a sequence of related probability distributions. SMC is most commonly used to solve filtering in state-space models (SSM, Chapter 29), but it can also be applied to other problems, such as sampling from a static (but possibly multi-modal) distribution, or for sampling rare events from some process.

Our presentation is based on the excellent tutorial [NLS19], and differs from traditional presentations, such as [Aru+02], by emphasizing the fact that we are sampling sequences of related variables, not just computing the filtering distribution of an SSM. This more general perspective will let us tackle static estimation problems, as we will see. For another good introduction to SMC, see [DJ11]. For a more formal (measure theoretic) treatment of SMC, using the **Feynman-Kac** formalism, see [CP20b].

13.1.1 Problem statement

In SMC, the goal is to sample from a sequence of related distributions of the form

$$\pi_t(\mathbf{z}_{1:t}) = \frac{1}{Z_t} \tilde{\gamma}_t(\mathbf{z}_{1:t}) \quad (13.1)$$

for $t = 1 : T$, where $\tilde{\gamma}_t$ is the unnormalized **target distribution**, π_t is the normalized version, and $\mathbf{z}_{1:t}$ are the random variables of interest. In some applications (e.g., filtering in an SSM), we care about each intermediate marginal distribution, $\pi_t(\mathbf{z}_t)$, for $t = 1 : T$; this is called **particle filtering**. (The word “particle” just means “sample”.) In other applications, we only care about the final distribution, $\pi_T(\mathbf{z}_T)$, and the intermediate steps are introduced just for computational reasons; this is called an **SMC sampler**. We briefly review both of these below, and go into more detail in later sections.

13.1.2 Particle filtering for state-space models

An important application of SMC is to sequential (online) inference (state estimation) in SSMs. As an example, consider a Markovian state-space model with the following joint distribution:

$$\pi_T(\mathbf{z}_{1:T}) \propto p(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{t=1}^T p(\mathbf{z}_t|\mathbf{z}_{t-1})p(\mathbf{y}_t|\mathbf{z}_t) \quad (13.2)$$

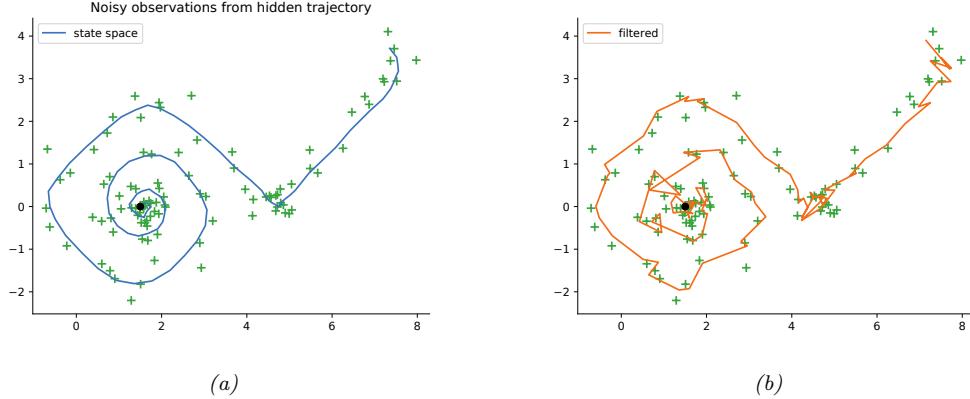


Figure 13.1: Illustration of particle filtering (using the dynamical prior as the proposal) applied to a 2d nonlinear dynamical system. (a) True underlying state and observed data. (b) PF estimate of the posterior mean. Generated by [bootstrap_filter_spiral.ipynb](#).

A common choice is to define the unnormalized target distribution at step t to be

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{s=1}^t p(\mathbf{z}_s|\mathbf{z}_{s-1})p(\mathbf{y}_s|\mathbf{z}_s) \quad (13.3)$$

Note that this is a distribution over an (ever growing) sequence of latent variables. However, we often only care about the most recent marginal of this distribution, in which case we just need to compute $\tilde{\gamma}_t(\mathbf{z}_t)$, which avoids having to store the full history.

For example, consider the following 2d nonlinear tracking problem (the same one as in Section 8.3.2.3):

$$\begin{aligned} p(\mathbf{z}_t|\mathbf{z}_{t-1}) &= \mathcal{N}(\mathbf{z}_t|f(\mathbf{z}_{t-1}), q\mathbf{I}) \\ p(\mathbf{y}_t|\mathbf{z}_t) &= \mathcal{N}(\mathbf{y}_t|\mathbf{z}_t, r\mathbf{I}) \\ f(\mathbf{z}) &= (z_1 + \Delta \sin(z_2), z_2 + \Delta \cos(z_1)) \end{aligned} \quad (13.4)$$

where Δ is the step size of the underlying continuous system, q is the variance of the system noise, and r is the variance of the observation noise. (We treat Δ , q , and r as fixed constants; see Supplementary Section 13.1.3 for a discussion of joint state and parameter estimation.) The true underlying state trajectory, and the corresponding noisy measurements, are shown in Figure 13.1a. The posterior mean estimate of the state, computed using 2000 samples in a simple form of SMC called the bootstrap filter (Section 13.2.3.1), is shown in Figure 13.1b.

Particle filtering can also be applied to **non-Markovian models**, where \mathbf{z}_t may depend on all the past hidden states, $\mathbf{z}_{1:t-1}$, and \mathbf{y}_t depends on the current \mathbf{z}_t and possibly also all the past hidden states, $\mathbf{z}_{1:t-1}$, and optionally the past observations, $\mathbf{y}_{1:t-1}$. In this case, the unnormalized target

distribution at step t is

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_1)p(\mathbf{y}_1|\mathbf{z}_1) \prod_{s=1}^t p(\mathbf{z}_s|\mathbf{z}_{1:s-1})p(\mathbf{y}_s|\mathbf{z}_{1:s}) \quad (13.5)$$

For example, consider a 1d Gaussian sequence model where the dynamics are first-order Markov, but the observations depend on the entire past sequence (this is example 1.2.1 from [NLS19]):

$$\begin{aligned} p(z_t|\mathbf{z}_{1:t-1}) &= \mathcal{N}(z_t|\phi z_{t-1}, q) \\ p(y_t|\mathbf{z}_{1:t}) &= \mathcal{N}(y_t|\sum_{s=1}^t \beta^{t-s} z_s, r) \end{aligned} \quad (13.6)$$

If we set $\beta = 0$, we get $p(y_t|\mathbf{z}_{1:t}) = \mathcal{N}(y_t|z_t, r)$ (where we define $0^0 = 1$), so the model becomes a linear-Gaussian SSM. As β gets larger, the dependence on the past increases, making the inference problem harder. (We will revisit this example below.)

13.1.3 SMC samplers for static parameter estimation

Now consider the problem of parameter estimation from a fixed dataset, $\mathcal{D} = \{\mathbf{y}_n : n = 1 : N\}$. We suppose the observations are conditionally iid, so the posterior has the form $p(\mathbf{z}|\mathcal{D}) \propto p(\mathbf{z}) \prod_{n=1}^N p(\mathbf{y}_n|\mathbf{z})$, where \mathbf{z} is the unknown parameter. It is not immediately obvious how to approximate $p(\mathbf{z}|\mathcal{D})$ using SMC, since we just have one distribution. However, we can convert this into a sequential inference problem in several different ways. One approach, known as **data tempering**, defines the (marginal) target distribution at step t as $\tilde{\gamma}_t(\mathbf{z}_t) = p(\mathbf{z}_t)p(\mathbf{y}_{1:t}|\mathbf{z}_t)$. In this case, the number of time steps T is the same as the number of data samples, N . Another approach, known as **likelihood tempering**, defines the (marginal) target distribution at step t as $\tilde{\gamma}_t(\mathbf{z}_t) = p(\mathbf{z}_t)p(\mathcal{D}|\mathbf{z}_t)^{\tau_t}$, where $0 = \tau_t < \dots < \tau_T = 1$ is a temperature parameter. In this case, the number of steps T depends on how quickly we anneal the distribution from the initial prior $p(\mathbf{z}_1)$ to the final target $p(\mathbf{z}_T)p(\mathcal{D}|\mathbf{z}_T)$.

Once we have defined the marginal target distributions $\tilde{\gamma}_t(\mathbf{z}_t)$, we need a way to expand this to a joint target distribution over a *sequence* of variables, $\tilde{\gamma}_t(\mathbf{z}_{1:t})$, so the distributions become connected to each other. We explain how to do this in Section 13.6. We can then treat the model as an SSM and apply particle filtering. At the end, we extract the final joint target distribution, $\tilde{\gamma}_T(\mathbf{z}_{1:T}) = p(\mathbf{z}_{1:T})p(\mathcal{D}|\mathbf{z}_T)$, from which we can compute the marginal target distribution $\tilde{\gamma}_T(\mathbf{z}_T) = p(\mathbf{z}_T, \mathcal{D})$, from which we can get the posterior $p(\mathbf{z}|\mathcal{D})$ by normalizing. We give the details in Section 13.6.

13.2 Particle filtering

In this section, we cover the basics of SMC for state space models, culminating in a method known as the **particle filter**.

13.2.1 Importance sampling

We start by reviewing the self-normalized importance sampling method (**SNIS**), which is the foundation of the particle filter. (See also Section 11.5.)

Suppose we are interested in estimating the expectation of some function φ_t with respect to a target distribution π_t , which we denote by

$$\pi_t(\varphi) \triangleq \mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] = \int \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{Z_t} \varphi_t(\mathbf{z}_{1:t}) d\mathbf{z}_{1:t} \quad (13.7)$$

where $Z_t = \int \tilde{\gamma}_t(\mathbf{z}_{1:t}) d\mathbf{z}_{1:t}$. Suppose we use SNIS with proposal $q_t(\mathbf{z}_{1:t})$. We then get the following approximation:

$$\pi_t(\varphi) \approx \frac{1}{\hat{Z}_t} \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t(\mathbf{z}_{1:t}^i) \varphi_t(\mathbf{z}_{1:t}^i) \quad (13.8)$$

where $\mathbf{z}_{1:t}^i \stackrel{\text{iid}}{\sim} q_t$ are independent samples from the proposal, \tilde{w}_t^i are the **unnormalized weights** defined by

$$\tilde{w}_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{q_t(\mathbf{z}_{1:t}^i)} \quad (13.9)$$

and \hat{Z}_t is the approximate normalization constant defined by

$$\hat{Z}_t \triangleq \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i \quad (13.10)$$

To simplify notation, let us define the **normalized weights** by

$$W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j} \quad (13.11)$$

Then we can write

$$\mathbb{E}_{\pi_t} [\varphi_t(\mathbf{z}_{1:t})] \approx \sum_{i=1}^{N_s} W_t^i \varphi_t(\mathbf{z}_{1:t}^i) \quad (13.12)$$

Alternatively, instead of computing the expectation of a specific target function, we can just approximate the target distribution itself, using a sum of weighted samples:

$$\pi_t(\mathbf{z}_{1:t}) \approx \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i) \triangleq \hat{\pi}_t(\mathbf{z}_{1:t}) \quad (13.13)$$

The problem with importance sampling when applied in the context of sequential models is that the dimensionality of the state space is very large, and increases with t . This makes it very hard to define a good proposal that covers the high probability regions, resulting in most samples getting negligible weight. In the sections below, we discuss solutions to this problem.

13.2.2 Sequential importance sampling

In this section, we discuss **sequential importance sampling** or **SIS**, in which the proposal has the following autoregressive structure:

$$q_t(\mathbf{z}_{1:t}) = q_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1}) \quad (13.14)$$

We can obtain samples from $q_{t-1}(\mathbf{z}_{1:t-1})$ by reusing the $\mathbf{z}_{1:t-1}^i$ samples, which we then extend by one step by sampling from the conditional $q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1}^i)$. We can think of this as “growing” the chain (sequence of states). The unnormalized weights can be computed recursively as follows:

$$\tilde{w}_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{q_t(\mathbf{z}_{1:t})} = \frac{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})q_{t-1}(\mathbf{z}_{1:t-1})} \quad (13.15)$$

$$= \frac{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{q_{t-1}(\mathbf{z}_{1:t-1})} \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.16)$$

$$= \tilde{w}_{t-1}(\mathbf{z}_{1:t-1}) \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.17)$$

The ratio factors are sometimes called the **incremental importance weights**:

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.18)$$

See Algorithm 13.1 for pseudocode for the resulting SIS algorithm. (In practice we compute the weights in log-space, and convert back using the log-sum-exp trick.)

Note that, in the special case of state space models, the weight computation can be further simplified. In particular, suppose we have

$$\tilde{\gamma}_t(\mathbf{z}_{1:t}) = p(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) = p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})p(\mathbf{z}_{1:t-1}, \mathbf{y}_{1:t-1}) \quad (13.19)$$

$$= p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}) \quad (13.20)$$

Then the incremental weight is given by

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} = \frac{p(\mathbf{y}_t|\mathbf{z}_{1:t})p(\mathbf{z}_t|\mathbf{z}_{1:t-1})}{q_t(\mathbf{z}_t|\mathbf{z}_{1:t-1})} \quad (13.21)$$

Unfortunately SIS suffers from a problem known as **weight degeneracy** or **particle impoverishment**, in which most of the weights become very small (near zero), so the posterior ends up being approximated by a single particle. This is illustrated in Figure 13.2a, where we apply SIS to the non-Markovian example in Equation (13.6) using $N_s = 5$ particles. The reason for degeneracy is that each particle has to “explain” (generate) the entire sequence of observations. Each sequence of guessed states becomes increasingly improbable over time, due to the product of likelihood terms, and the differences between the weights of each hypothesis will grow exponentially. Of course, there has to be a best sequence amongst the set of candidates, so when we normalize the weights, the best one will get weight 1 and the rest will get weight 0. But this is a waste of most of the particles. We discuss a solution to this in Section 13.2.3.

Algorithm 13.1: Sequential importance sampling (SIS)

```

1 Initialization:  $\mathbf{z}_1^i \sim q_1(\mathbf{z}_1)$ ,  $\tilde{w}_1^i = \frac{\tilde{\gamma}_1(\mathbf{z}_1^i)}{q_1(\mathbf{z}_1^i)}$ ,  $W_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$ ,  $\hat{\pi}_1(\mathbf{z}_1) = \sum_{i=1}^{N_s} W_1^i \delta(\mathbf{z}_1 - \mathbf{z}_1^i)$ 
2 for  $t = 2 : T$  do
3   for  $i = 1 : N_s$  do
4     Sample  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
5     Compute incremental weight  $\alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
6     Compute unnormalized weight  $\tilde{w}_t^i = \tilde{w}_{t-1}^i \alpha_t^i$ 
7     Compute normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
8   Compute MC posterior  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

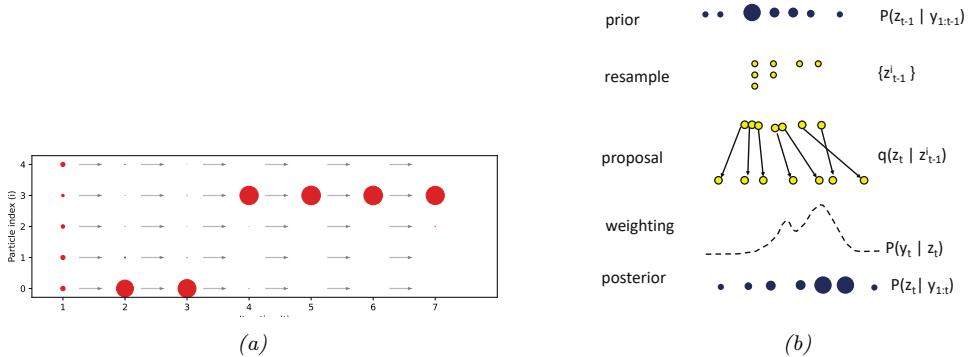


Figure 13.2: (a) Illustration of weight degeneracy for SIS applied to the model in Equation (13.6). with parameters $(\phi, q, \beta, r) = (0.9, 10.0, 0.5, 1.0)$. We use $T = 6$ steps and $N_s = 5$ samples. We see that as t increases, almost all the probability mass concentrates on particle 3. Generated by [sis_vs_smcmc.ipynb](#). Adapted from Figure 2 of [NLS19]. (b) Illustration of the bootstrap particle filtering algorithm.

13.2.3 Sequential importance sampling with resampling

In this section, we describe **sequential importance sampling with resampling (SISR)**. The basic idea is this: instead of “growing” all of the old particle sequences by one step, we first select the N_s “fittest” particles, by sampling from the old posterior, and then we let these survivors grow by one step.

In more detail, at step t , we sample from

$$q_t^{\text{SISR}}(\mathbf{z}_{1:t}) = \hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.22)$$

where $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1})$ is the previous weighted posterior approximation. By contrast, in SIS, we sample from

$$q_t^{\text{SIS}}(\mathbf{z}_{1:t}) = q_{t-1}^{\text{SIS}}(\mathbf{z}_{1:t-1}) q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.23)$$

Algorithm 13.2: Sequential importance sampling with resampling (SISR)

```

1 Initialization:  $\mathbf{z}_1^i \sim q_1(\mathbf{z}_1)$ ,  $\tilde{w}_1^i = \frac{\tilde{\gamma}_1(\mathbf{z}_1^i)}{q_1(\mathbf{z}_1^i)}$ ,  $W_1^i = \frac{\tilde{w}_1^i}{\sum_j \tilde{w}_1^j}$ ,  $\hat{\pi}_1(\mathbf{z}_1) = \sum_{i=1}^{N_s} W_1^i \delta(\mathbf{z}_1 - \mathbf{z}_1^i)$ 
2 for  $t = 2 : T$  do
3   Compute ancestors  $\mathbf{a}_{t-1}^{1:N_s} = \text{resample}(\tilde{w}_{t-1}^{1:N_s})$ 
4   Select  $\mathbf{z}_{t-1}^{1:N_s} = \text{permute}(\mathbf{a}_{t-1}^{1:N_s}, \mathbf{z}_{t-1}^{1:N_s})$ 
5   Reset unnormalized weights  $\tilde{w}_{t-1}^{1:N_s} = 1/N_s$ 
6   for  $i = 1 : N_s$  do
7     Sample  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
8     Compute unnormalized weight  $\tilde{w}_t^i = \alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
9     Compute normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
10    Compute MC posterior  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

We can sample from Equation (13.22) in two steps. First we **resample** N_s samples from $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1})$ to get a *uniformly weighted* set of new samples $\mathbf{z}_{1:t-1}^i$. (See Section 13.2.4 for details on how to do this.) Then we extend each sample using $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$, and concatenate \mathbf{z}_t^i to $\mathbf{z}_{1:t-1}^i$.

After making a proposal, we compute the unnormalized weights. We use the standard SNIS method, except we “pretend” that the proposal is given by $\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)$ even though we used $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}^i) q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)$. The intuitive reason why this is valid is because the previous weighted approximation, $\hat{\pi}_{t-1}(\mathbf{z}_{1:t-1}^i)$, was an unbiased estimate of the previous target distribution, $\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1})$. (See e.g., [CP20b] for more theoretical details.) We then compute the unnormalized weights, which are the same as the incremental weights, since the resampling step sets $\tilde{w}_{t-1}^i = 1$. We then normalize these weights and compute the new approximation to the target posterior $\hat{\pi}_t(\mathbf{z}_{1:t})$. See Algorithm 13.2 for the pseudocode.

13.2.3.1 Bootstrap filter

We now consider a special case of SISR, in which the model is an SSM, and the proposal distribution is equal to the dynamical prior:

$$q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \quad (13.24)$$

In this case, the corresponding incremental weight in Equation (13.21) simplifies to

$$\alpha_t(\mathbf{z}_{1:t}) = \frac{p(\mathbf{y}_t | \mathbf{z}_{1:t}) p(\mathbf{z}_t | \mathbf{z}_{1:t-1})}{q(\mathbf{z}_t | \mathbf{z}_{1:t-1})} = \frac{p(\mathbf{y}_t | \mathbf{z}_t) p(\mathbf{z}_t | \mathbf{z}_{t-1})}{p(\mathbf{z}_t | \mathbf{z}_{t-1})} = p(\mathbf{y}_t | \mathbf{z}_{1:t}) \quad (13.25)$$

This special case is called the **bootstrap filter** [Gor93] or the **survival of the fittest** algorithm [KKR95]. (In the computer vision literature, this is called the **condensation** algorithm, which stands for “conditional density propagation” [IB98].) See Figure 13.2b for an illustration of how this algorithm works, and Figure 13.1b for some sample results on real data.

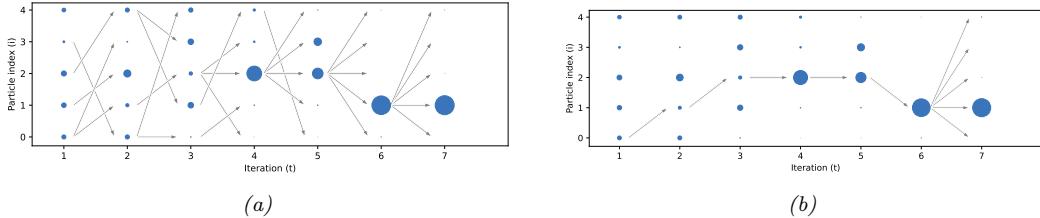


Figure 13.3: (a) Illustration of diversity of samples in SMC applied to the model in Equation (13.6). (b) Illustration of the path degeneracy problem. Generated by `sis_vs_smc.ipynb`. Adapted from Figure 3 of [NLS19].

The bootstrap filter is useful for models where we can sample from the dynamics, but cannot evaluate the transition model pointwise. This occurs in certain implicit dynamical models, such as those defined using differential equations (see e.g., [IBK06]); such models are often used in epidemiology. However, in general it is much more efficient to use proposals that take the current evidence \mathbf{y}_t into account. We discuss ways to approximate such “locally optimal” proposals in Section 13.3.

13.2.3.2 Path degeneracy problem

In Figure 13.3a we show how particle filtering can result in a much more diverse set of active particles, with more balanced weights when applied to the non-Markovian example in Equation (13.6).

While particle filtering does not suffer from weight degeneracy, it does suffer from another problem known as **path degeneracy**. This refers to the fact that the number of particles that “survive” (have non-negligible weight) over many steps may drop rapidly over time, resulting in a loss of diversity when we try to represent the distribution over the past. We illustrate this in Figure 13.3b, where we only include arrows for samples that have been resampled at each step up until the final step. We see that we have $N_s = 5$ identical copies of \mathbf{z}_1^1 in the final set of surviving sequences. (The time at which all the paths meet at a common ancestor, when tracing backwards in time, is known as the **coalescence** time.) We discuss some ways to ameliorate this issue in Section 13.2.4 and Section 13.2.5.

13.2.3.3 Estimating the normalizing constant

We can use particle filtering to approximate the normalization constant $Z_T = p(\mathbf{y}_{1:T}) = \prod_{t=1}^T p(\mathbf{y}_t | \mathbf{y}_{1:t-1})$ as follows:

$$\hat{Z}_T = \prod_{t=1}^T \hat{Z}_t \quad (13.26)$$

where, from Equation (13.10), we have

$$\hat{Z}_t = \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{w}_t^i = \hat{Z}_{t-1} \left(\widehat{Z_t / Z_{t-1}} \right) \quad (13.27)$$

where

$$\widehat{Z_t/Z_{t-1}} = \frac{\sum_{i=1}^{N_s} \tilde{w}_t^i}{\sum_{i=1}^{N_s} \tilde{w}_{t-1}^i} \quad (13.28)$$

This estimate of the marginal likelihood is very useful for tasks such as parameter estimation.

13.2.4 Resampling methods

Importance sampling gives a weighted set of particles, $\{(W_t^i, \mathbf{z}_t^i) : i = 1 : N\}$, which we can use to approximate posterior expectations using

$$\mathbb{E}[f(\mathbf{z}_t) | \mathbf{y}_{1:t}] \approx \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.29)$$

Suppose we sample a single index $A \in \{1, \dots, N\}$ with probabilities (W_t^1, \dots, W_t^N) . Then the expected value evaluated at this index is

$$\mathbb{E}[f(\mathbf{z}_t^A) | \mathbf{y}_{1:t}] = \sum_{i=1}^N p(A=i) f(\mathbf{z}_t^i) = \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.30)$$

If we sample N indices independently and compute their average, we get

$$\mathbb{E}[f(\mathbf{z}_t) | \mathbf{y}_{1:t}, A_{1:N}] \approx \frac{1}{N} \sum_{i=1}^N f(\mathbf{z}_t^{A_i}) \quad (13.31)$$

which is a standard unweighted Monte Carlo estimate, with weights $W_t^i = 1/N$. Averaging over the indices gives

$$\mathbb{E}_{A_{1:N}} \left[\frac{1}{N} \sum_{i=1}^N f(\mathbf{z}_t^{A_i}) \right] = \sum_{i=1}^N W_t^i f(\mathbf{z}_t^i) \quad (13.32)$$

Thus using the output from the resampling procedure — which drops particles with low weight, and duplicates particles with high weight — will give the same result in expectation as the original weighted estimate. However, to reduce the variance of the method, we need to pick the resampling method carefully, as we discuss below.

13.2.4.1 Inverse cdf

Most of the common resampling methods work as follows. First we form the cumulative distribution from the weights $W^{1:N}$, as illustrated by the staircase in Figure 13.4. (We drop the t index for brevity.) Then, given a set of N uniform random variables, $U^i \sim \text{Unif}(0, 1)$, we check to see which bin (interval) U^i lands in; if it falls in bin a , we return index a , i.e., sample i gets mapped to index a if

$$\sum_{j=1}^{a-1} W^j \leq U^i < \sum_{j=1}^a W^j \quad (13.33)$$

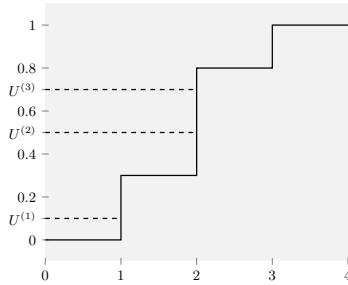


Figure 13.4: Illustration of how to sample from the empirical cdf $P(x) = \sum_{n=1}^N W^n \mathbb{I}(x \geq n)$ shown in black. The height of step n is W_n . If U^m picks step n , then we set the ancestor of m to be n , i.e., $A^m = n$. In this example, $A^{1:3} = (1, 2, 2)$. Adapted from Figure 9.3 of [CP20b].

It would seem that each index would take $O(N)$ time to compute, for a total time of $O(N^2)$, but if the U^i are ordered from smallest to largest, we can implement it in $O(N)$ time. We denote this function $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$. See Listing 13.1 for some JAX code.¹

Listing 13.1: Sampling from an ordered inverse CDF

```
def icdf(weights, u):
    n = weights.shape[0]
    cumsum = jnp.cumsum(weights)
    idx = jnp.searchsorted(cumsum, u)
    return jnp.clip(idx, 0, n - 1)
```

13.2.4.2 Multinomial resampling

In **multinomial resampling**, we set $\mathbf{U}^{1:N}$ to be an ordered set of N samples from the uniform distribution. We then compute the ancestor indices using $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.

Although this is a simple method, it can introduce a lot of variance into the representation of the distribution. For example, suppose all the weights are equal, $W^n = 1/N$. Let $\mathcal{W}^n = \sum_{m=1}^N \mathbb{I}(A^m = n)$ be the number of “offspring” for particle n (i.e., the number of times this particle is chosen in the resampling step). We have $\mathcal{W}^n \sim \text{Bin}(N, 1/N)$, so $P(\mathcal{W}^n = 0) = (1 - 1/N)^N \approx e^{-1} \approx 0.37$. So there is a 37% chance that any given particle will disappear even though they all had the same initial weight. In the sections below, we discuss some **low variance resampling** methods.

13.2.4.3 Stratified resampling

A simple approach to improve on multinomial resampling is to use **stratified resampling**, in which we divide the unit interval into N_s strata, $(0, 1/N_s)$, $(1/N_s, 2/N_s)$, up to $(1 - 1/N_s, 1)$. We then generate

$$U^i \sim \text{Unif}((i - 1)/N_s, i/N_s) \tag{13.34}$$

and compute $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.²

1. Modified from <https://github.com/blackjax-devs/blackjax/blob/main/blackjax/smc/resampling.py>.
2. To compute the $\mathbf{U}^{1:N}$, we can use $v = \text{jnp.uniform(rngkey, (n,))}$ and $u = (\text{jnp.arange}(n) + v) / n$.

13.2.4.4 Systematic resampling

We can further reduce the variance by forcing all the samples to be deterministically generated from a shared random source, $u \sim \text{Unif}(0, 1)$, by computing

$$U^i = \frac{i - 1}{N_s} + \frac{u}{N_s} \quad (13.35)$$

We then compute $\mathbf{A}_{1:N} = \text{icdf}(\mathbf{W}^{1:N}, \mathbf{U}^{1:N})$.³

13.2.4.5 Comparison

It can be proved that all of the above methods are unbiased. Empirically it seems that systematic resampling is lower variance than other methods [HSG06], although stratified resampling, and the more complex method of [GCW19], have better theoretical properties. Multinomial resampling is not recommended, since it has provably higher variance than the other methods.

13.2.5 Adaptive resampling

The resampling step can result in loss of diversity, since each ancestor may generate multiple children, and some may generate no children, since the ancestor indices A_t^n are sampled independently; this is the path degeneracy problem mentioned above. On the other hand, if we never resample, we end up with SIS, which suffers from weight degeneracy (particles with negligible weight). A compromise is to use **adaptive resampling**, in which we resample whenever the **effective sample size** or **ESS** drops below some minimum, such as $N/2$. A common way to define the ESS is as follows:⁴

$$\text{ESS}(W^{1:N}) = \frac{1}{\sum_{n=1}^N (W^n)^2} \quad (13.36)$$

Alternatively we can compute the ESS using the unnormalized weights:

$$\text{ESS}(\tilde{w}^{1:N}) = \frac{\left(\sum_{n=1}^N \tilde{w}^n\right)^2}{\sum_{n=1}^N (\tilde{w}^n)^2} \quad (13.37)$$

Note that if we have k weights with $\tilde{w}^n = 1$ and $N - k$ weights with $\tilde{w}^n = 0$, then the ESS is k ; thus ESS is between 1 and N .

The pseudocode for SISR with adaptive resampling is given in Algorithm 13.3. (We use the notation of [Law+22, App. B], in which we first sample new extensions of the sequences, and then optionally resample the sequences at the end of each step.)

13.3 Proposal distributions

The efficiency of PF is crucially dependent on the quality of the proposal distribution. We discuss some options below.

3. To compute the $\mathbf{U}^{1:N}$, we can use $v = \text{jnp.uniform(rngkey, ())}$ and $u = (\text{jnp.arange}(n) + v) / n$.

4. Note that the ESS used in SMC is different than the ESS used in MCMC (Section 12.6.3); the latter takes into account auto-correlation of the MCMC samples.

Algorithm 13.3: SISR with adaptive resampling (generic SMC)

```

1 Initialization:  $\tilde{w}_0^{1:N_s} = 1$ ,  $\hat{Z}_0 = 1$ 
2 for  $t = 1 : T$  do
3   for  $i = 1 : N_s$  do
4     Sample particle  $\mathbf{z}_t^i \sim q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i)$ 
5     Compute incremental weight  $\alpha_t^i = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t}^i)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{1:t-1}^i)q_t(\mathbf{z}_t^i | \mathbf{z}_{1:t-1}^i)}$ 
6     Compute unnormalized weight  $\tilde{w}_t^i = \tilde{w}_{t-1}^i \alpha_t^i$ 
7     Estimate normalization constant:  $\widehat{Z_t / Z_{t-1}} = \frac{\sum_{i=1}^{N_s} \tilde{w}_t^i}{\sum_{i=1}^{N_s} \tilde{w}_{t-1}^i}$ ,  $\hat{Z}_t = \hat{Z}_{t-1} (\widehat{Z_t / Z_{t-1}})$ 
8     if  $ESS(\tilde{w}_t^{1:N_s}) < ESS_{min}$  then
9       Compute ancestors  $\mathbf{a}_t^{1:N_s} = \text{resample}(\tilde{w}_t^{1:N_s})$ 
10      Select  $\mathbf{z}_t^{1:N_s} = \text{permute}(\mathbf{a}_t^{1:N_s}, \mathbf{z}_t^{1:N_s})$ 
11      Reset unnormalized weights  $\tilde{w}_t^{1:N_s} = 1/N_s$ 
12      Compute normalized weights  $W_t^i = \frac{\tilde{w}_t^i}{\sum_j \tilde{w}_t^j}$  for  $i = 1 : N_s$ 
13      Compute MC posterior  $\hat{\pi}_t(\mathbf{z}_{1:t}) = \sum_{i=1}^{N_s} W_t^i \delta(\mathbf{z}_{1:t} - \mathbf{z}_{1:t}^i)$ 

```

13.3.1 Locally optimal proposal

We define the (one-step) **locally optimal proposal distribution** $q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1})$ to be the one that minimizes

$$D_{\text{KL}}(\pi_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_{1:t})) \quad (13.38)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [\log \{\pi_{t-1}(\mathbf{z}_{1:t-1})q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})\} - \log \pi_t(\mathbf{z}_{1:t})] \quad (13.39)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [\log q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) - \log \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1})] + \text{const} \quad (13.40)$$

$$= \mathbb{E}_{\pi_{t-1}q_t} [D_{\text{KL}}(q_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) \| \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}))] + \text{const} \quad (13.41)$$

The KL is minimized by choosing

$$q_t^*(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{1:t})}{\tilde{\gamma}_t(\mathbf{z}_{1:t-1})} \quad (13.42)$$

where $\tilde{\gamma}_t(\mathbf{z}_{1:t-1}) = \int \tilde{\gamma}_t(\mathbf{z}_{1:t}) d\mathbf{z}_t$ is the probability of the past sequence under the current target distribution.

Note that the subscript t specifies the t 'th distribution, so in the context of SSMs, we have $\pi_t(\mathbf{z}_t | \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t | \mathbf{z}_{1:t-1}, \mathbf{y}_{1:t})$. Thus we see that when proposing \mathbf{z}_t , we should condition on all the data, including the most recent observation, \mathbf{y}_t ; this is called a **guided particle filter**, and will be better than the bootstrap filter, which proposes from the prior.

In general, it is intractable to compute the locally optimal proposal, so we consider various approximations below.

13.3.2 Proposals based on the extended and unscented Kalman filter

One way to approximate the locally optimal proposal distribution is based on the extended Kalman filter (Section 8.3.2) or the unscented Kalman filter (Section 13.3.2), which gives rise to the **extended particle filter** [DGA00] and **unscented particle filter** [Mer+00] respectively. To explain these methods, we follow the presentation of [NLS19, p36]. As usual, we assume the dynamical system can be written as $\mathbf{z}_t = \mathbf{f}(\mathbf{z}_{t-1}) + \mathbf{q}_t$ and $\mathbf{y}_t = \mathbf{h}(\mathbf{z}_t) + \mathbf{r}_t$, where \mathbf{q}_t is the system noise and \mathbf{r}_t is the observation noise. The EKF and UKF approximations assume that the joint distribution over neighboring time steps, given the i 'th history, is Gaussian:

$$p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{z}_{1:t-1}^i) \approx \mathcal{N}\left(\begin{pmatrix} \mathbf{z}_t \\ \mathbf{y}_t \end{pmatrix} | \hat{\boldsymbol{\mu}}^i, \hat{\boldsymbol{\Sigma}}^i\right) \quad (13.43)$$

where

$$\hat{\boldsymbol{\mu}}^i = \begin{pmatrix} \hat{\boldsymbol{\mu}}_z^i \\ \hat{\boldsymbol{\mu}}_y^i \end{pmatrix}, \hat{\boldsymbol{\Sigma}}^i = \begin{pmatrix} \hat{\boldsymbol{\Sigma}}_{zz}^i & \hat{\boldsymbol{\Sigma}}_{zy}^i \\ \hat{\boldsymbol{\Sigma}}_{yz}^i & \hat{\boldsymbol{\Sigma}}_{yy}^i \end{pmatrix} \quad (13.44)$$

(See Section 8.5.1 for details.)

The EKF and UKF compute $\hat{\boldsymbol{\mu}}^i$ and $\hat{\boldsymbol{\Sigma}}^i$ differently. In the EKF, we linearize \mathbf{f} and \mathbf{h} , and assume the noise terms are Gaussian. We then compute $p(\mathbf{z}_t, \mathbf{y}_t | \mathbf{z}_{1:t-1}^i)$ exactly for this linearized model (see Section 8.3.1). In the UKF, we propagate sigma points through \mathbf{f} and \mathbf{h} , and approximate the resulting means and covariances using the unscented transform, which can be more accurate (see Section 8.4). Once we have computed $\hat{\boldsymbol{\mu}}^i$ and $\hat{\boldsymbol{\Sigma}}^i$, we can use standard rules for Gaussian conditioning to compute the approximate proposal as follows:

$$q(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i, \mathbf{y}_t) \approx \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_t^i, \boldsymbol{\Sigma}_t^i) \quad (13.45)$$

$$\boldsymbol{\mu}_t^i = \hat{\boldsymbol{\mu}}_z^i + \hat{\boldsymbol{\Sigma}}_{zy}^i (\hat{\boldsymbol{\Sigma}}_{yy}^i)^{-1} (\mathbf{y}_t - \hat{\boldsymbol{\mu}}_y^i) \quad (13.46)$$

$$\boldsymbol{\Sigma}_t^i = \hat{\boldsymbol{\Sigma}}_{zz}^i - \hat{\boldsymbol{\Sigma}}_{zy}^i (\hat{\boldsymbol{\Sigma}}_{yy}^i)^{-1} \hat{\boldsymbol{\Sigma}}_{yz}^i \quad (13.47)$$

Note that the linearization (or sigma point) approximation needs to be performed for each particle separately.

13.3.3 Proposals based on the Laplace approximation

To handle non-Gaussian likelihoods in an SSM, we can use the Laplace approximation (Section 7.4.3), as suggested in [DGA00]. In particular, consider an SSM with linear-Gaussian latent dynamics and a GLM likelihood. At each step, we compute the maximum $\mathbf{z}_t^* = \text{argmax} \log p(\mathbf{y}_t | \mathbf{z}_t)$ as step t (e.g., using Newton-Raphson), and then approximate the likelihood using

$$p(\mathbf{y}_t | \mathbf{z}_t) \approx \mathcal{N}(\mathbf{z}_t | \mathbf{z}_t^*, -\mathbf{H}_t^*) \quad (13.48)$$

where \mathbf{H}_t^* is the Hessian of the log-likelihood at the mode. We now compute $p(\mathbf{z}_t | \mathbf{z}_{t-1}^i, \mathbf{y}_t)$ using the update step of the Kalman filter, using the same equations as in Section 13.3.2. This combination is called the **Laplace Gaussian filter** [Koy+10]. We give an example in Section 13.3.3.1.

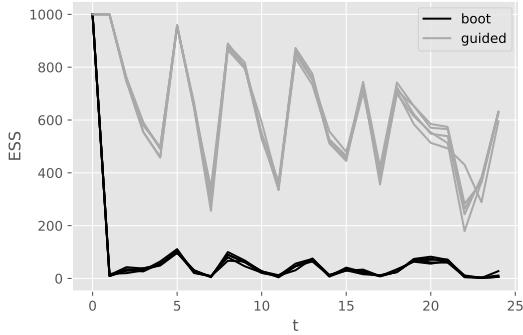


Figure 13.5: Effective sample size at each step for the bootstrap particle filter and a guided particle filter for a Gaussian SSM with Poisson likelihood. Adapted from Figure 10.4 of [CP20b]. Generated by [pf_guided_neural_decoding.ipynb](#).

13.3.3.1 Example: neural decoding

In this section, we give an example where we apply the Laplace approximation to an SSM with linear-Gaussian dynamics and a Poisson likelihood. The application arises from neuroscience. In particular, assume we record the **neural spike trains** as a monkey moves its hand around in space. Let $\mathbf{z}_t \in \mathbb{R}^6$ represent the 3d location and velocity of the hand. We model the dynamics of the hand using a simple Brownian random walk model [CP20b, p157]:

$$\begin{pmatrix} z_t(i) \\ z_t(i+3) \end{pmatrix} | \mathbf{z}_{t-1} \sim \mathcal{N}_2 \left(\begin{pmatrix} 1 & \Delta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} z_{t-1}(i) \\ z_{t-1}(i+3) \end{pmatrix}, \sigma^2 \mathbf{Q} \right), \quad i = 1 : 3 \quad (13.49)$$

where the covariance of the noise is given by the following, assuming a discretization step of Δ :

$$\mathbf{Q} = \begin{pmatrix} \Delta^3/3 & \Delta^2/2 \\ \Delta^2/2 & \Delta \end{pmatrix} \quad (13.50)$$

We assume the k 'th observation at time t is the number of spikes for neuron k in this sensing interval:

$$p(y_t(k) | \mathbf{z}_t) = \text{Poi}(\lambda_k(\mathbf{z}_t)) \quad (13.51)$$

$$\log \lambda_k(\mathbf{z}_t) = \alpha_k + \beta_k^\top \mathbf{z}_t \quad (13.52)$$

Our goal is to compute $p(\mathbf{z}_t | \mathbf{y}_{1:t})$, which lets us infer the position of the hand from the neural code. (Apart from its value for furthering basic science, this can be useful for applications such as helping disabled people control their arms using “mind control”.)

To illustrate this, we sample a synthetic dataset from the model, to simulate a “monkey” moving its arm for $T = 25$ time steps; this generates $K = 50$ neuronal counts per time step. We then apply particle filtering to this dataset (using the true model), using either the bootstrap filter (i.e., proposal is the random walk prior) or the guided filter (i.e., proposal is the Laplace approximation mentioned above). In Figure 13.5, we see that the effective sample size of the guided filter is much higher than for the bootstrap filter.

13.3.4 Proposals based on SMC (nested SMC)

It is possible to use SMC as a subroutine to compute a proposal distribution for SMC: at each step t , for each particle i , we run an SMC algorithm where the target distribution is the optimal proposal, $p(\mathbf{z}_t | \mathbf{z}_{1:t-1}^i, \mathbf{y}_{1:t})$. This is called **nested SMC** [NLS15; NLS19].

This method can approximate the locally optimal proposal arbitrarily well, since it does not make any limiting parametric assumptions. However, the method can be slow, although the inner SMC algorithm can be run in parallel for each outer sample [NLS15; NLS19].

13.4 Rao-Blackwellized particle filtering (RBPF)

In some models, we can partition the hidden variables into two kinds, \mathbf{m}_t and \mathbf{z}_t , such that we can analytically integrate out \mathbf{z}_t provided we know the values of $\mathbf{m}_{1:t}$. This means we only have to sample $\mathbf{m}_{1:t}$, and can represent $p(\mathbf{z}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t})$ parametrically. These hybrid particles are sometimes called **distributional particles** or **collapsed particles** [KF09a, Sec 12.4]. This combines techniques from particle filtering (Section 13.2) with deterministic methods such as Kalman filtering (Section 8.2.2).

The advantage of this approach is that we reduce the dimensionality of the space in which we are sampling, which reduces the variance of our estimate. This technique is known as **Rao-Blackwellized particle filtering** or **RBPF** for short. (See Section 11.6.2 for more details on Rao-Blackwellization.) In Section 13.4.1 we give an example of RBPF for inference in a switching linear dynamical systems. In Section 13.4.3 we illustrate RBPF for inference in the SLAM model for a mobile robot.

13.4.1 Mixture of Kalman filters

In this section, we consider the application of RBPF to a switching linear dynamical system (Section 29.9). This model has both continuous and discrete latent variables. This can be used to track a system that switches between discrete modes or operating regimes, represented by the discrete variable m_t .

For notational simplicity, we ignore the control inputs \mathbf{u}_t . Thus the model is given by

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, m_t = k) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}_k \mathbf{z}_{t-1}, \mathbf{Q}_k) \quad (13.53)$$

$$p(\mathbf{y}_t | \mathbf{z}_t, m_t = k) = \mathcal{N}(\mathbf{y}_t | \mathbf{H}_k \mathbf{z}_t, \mathbf{R}_k) \quad (13.54)$$

$$p(m_t = k | m_{t-1} = j) = A_{jk} \quad (13.55)$$

We let $\boldsymbol{\theta}_k = (\mathbf{F}_k, \mathbf{H}_k, \mathbf{Q}_k, \mathbf{R}_k, \mathbf{A}_{:,k})$ represent all the parameters for state k .

Exact inference is intractable, but if we sample the discrete variables, we can infer the continuous variables conditioned on the discretes exactly, making this a good candidate for RBPF. In particular, if we sample trajectories $\mathbf{m}_{1:t}^n$, we can apply a Kalman filter to each particle. This can be thought of as a **mixture of Kalman filters** [CL00]. The resulting belief state is represented by

$$p(\mathbf{z}_t, \mathbf{m}_t | \mathbf{y}_{1:t}) \approx \sum_{n=1}^N W_t^n \delta(\mathbf{m}_t - \mathbf{m}_t^n) \mathcal{N}(\mathbf{z}_t | \boldsymbol{\mu}_t^n, \boldsymbol{\Sigma}_t^n) \quad (13.56)$$

To derive the filtering algorithm, note that the full posterior at time t can be written as follows:

$$p(\mathbf{m}_{1:t}, \mathbf{z}_{1:t} | \mathbf{y}_{1:t}) = p(\mathbf{z}_{1:t} | \mathbf{m}_{1:t}, \mathbf{y}_{1:t}) p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) \quad (13.57)$$

The second term is given by the following:

$$p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) \propto p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_{1:t} | \mathbf{y}_{1:t-1}) \quad (13.58)$$

$$= p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_t | \mathbf{m}_{1:t-1}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.59)$$

$$= p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) p(\mathbf{m}_t | \mathbf{m}_{t-1}) p(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t-1}) \quad (13.60)$$

Note that, unlike the case of standard particle filtering, we cannot write $p(\mathbf{y}_t | \mathbf{m}_{1:t}, \mathbf{y}_{1:t-1}) = p(\mathbf{y}_t | \mathbf{m}_t)$, since \mathbf{m}_t does not d-separate the past observations from \mathbf{y}_t , as is evident from Figure 29.25a.

Suppose we use the following recursive proposal distribution:

$$q(\mathbf{m}_{1:t} | \mathbf{y}_{1:t}) = q(\mathbf{m}_t | \mathbf{m}_{1:t-1}, \mathbf{y}_{1:t}) q(\mathbf{m}_{1:t-1} | \mathbf{y}_{1:t}) \quad (13.61)$$

Then we get the unnormalized importance weights

$$\tilde{w}_t^n \propto \frac{p(\mathbf{y}_t | m_t^n, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) p(m_t^n | m_{t-1}^n)}{q(m_t^n | \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t})} \tilde{w}_{t-1}^n \quad (13.62)$$

As a special case, suppose we propose from the prior, $q(m_t | m_{t-1}^n, \mathbf{y}_{1:t}) = p(m_t | m_{t-1}^n)$. If we sample discrete state k , the weight update becomes

$$\tilde{w}_t^n \propto \tilde{w}_{t-1}^n p(\mathbf{y}_t | m_t^n = k, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \tilde{w}_{t-1}^n L_{tk}^n \quad (13.63)$$

where

$$L_{tk}^n = p(\mathbf{y}_t | m_t = k, \mathbf{m}_{1:t-1}^n, \mathbf{y}_{1:t-1}) = \int p(\mathbf{y}_t | m_t = k, \mathbf{z}_t) p(\mathbf{z}_t | m_t = k, \mathbf{y}_{1:t-1}, \mathbf{m}_{1:t-1}^n) d\mathbf{z}_t \quad (13.64)$$

The quantity L_{tk}^n is the predictive density for the new observation \mathbf{y}_t conditioned on $m_t = k$ and the history of previous latents, $\mathbf{m}_{1:t-1}^n$. In the case of SLDS models, this can be computed using the normalization constant of the Kalman filter, Equation (8.35). The resulting algorithm is shown in Algorithm 13.4. The step marked ‘‘KFupdate’’ refers to the Kalman filter update equations in Section 8.2.2, and is applied to each particle separately.

Algorithm 13.4: One step of RBPF for SLDS using prior as proposal

```

1 for n = 1 : N do
2   k ~ p(m_t | m_{t-1}^n)
3   m_t^n := k
4   ( $\mu_t^n, \Sigma_t^n, L_{tk}^n$ ) = KFupdate( $\mu_{t-1}^n, \Sigma_{t-1}^n, \mathbf{y}_t, \theta_k$ )
5    $\tilde{w}_t^n = \tilde{w}_{t-1}^n L_{tk}^n$ 
6 Compute ESS = ESS( $\tilde{w}_t^{1:N_s}$ )
7 if ESS < ESS_min then
8    $a_t^{1:N} = \text{Resample}(\tilde{w}_t^{1:N})$ 
9   ( $\mathbf{m}_t^{1:N_s}, \mu_t^{1:N_s}, \Sigma_t^{1:N_s}$ ) = permute( $\mathbf{a}_t, \mathbf{m}_t^{1:N_s}, \mu_t^{1:N_s}, \Sigma_t^{1:N_s}$ )
10   $\tilde{w}_t^n = 1/N_s$ 

```

13.4.1.1 Improvements

An improved version of the algorithm can be developed based on the fact that we are sampling a discrete state space. At each step, we propagate each of the N old particles through all K possible transition models. We then compute the weight for all NK new particles, and sample from this to get the final set of N particles. This latter step can be done using the **optimal resampling** method of [FC03], which will stochastically select the particles with the largest weight, while also ensuring the result is an unbiased approximation. In addition, this approach ensures that we do not have duplicate particles, which is wasteful and unnecessary when the state space is discrete.

13.4.2 Example: tracking a maneuvering object

In this section we give an example of RBPF for an SLDS from [DGK01]. Our goal is to track an object that has the following motion model:

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, m_t = k) = \mathcal{N}(\mathbf{z}_t | \mathbf{F}\mathbf{z}_{t-1} + \mathbf{b}_k, \mathbf{Q}) \quad (13.65)$$

where $\mathbf{z}_t = (x_{1t}, \dot{x}_{1t}, x_{2t}, \dot{x}_{2t})$ contains the 2d position and velocity. We define the observation matrix by $\mathbf{H} = \mathbf{I}$ and the observation covariance by $\mathbf{R} = 10 \text{ diag}(2, 1, 2, 1)$. We define the dynamics matrix by

$$\mathbf{F} = \begin{pmatrix} 1 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (13.66)$$

where $\Delta = 0.1$. We set the noise covariance to $\mathbf{Q} = 0.2\mathbf{I}$ and the input bias vectors for each state to $\mathbf{b}_1 = (0, 0, 0, 0)$, $\mathbf{b}_2 = (-1.225, -0.35, 1.225, 0.35)$ and $\mathbf{b}_3 = (1.225, 0.35, -1.225, -0.35)$. Thus the system will turn in different directions depending on the discrete state. The discrete state transition matrix is given by

$$\mathbf{A} = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix} \quad (13.67)$$

Figure 13.6a shows some observations, and the true state of the system, from a sample run, for 100 steps. The colors denote the discrete state, and the location of the symbol denotes the (x, y) location. The small dots represent noisy observations. Figure 13.6b shows the estimate of the state computed using RBPF with the optimal proposal with 1000 particles. In Figure 13.6c, we show the analogous estimate using the bootstrap filter, which does much worse.

In Figure 13.7a and Figure 13.7b, we show the posterior marginals of the (x, y) locations over time. In Figure 13.7c we show the true discrete state, and in Figure 13.7d we show the posterior marginal over discrete states. The overall state classification error rate is 29%, but it seems that occasionally misclassifying isolated time steps does not significantly hurt estimation of the continuous states, as we can see from Figure 13.6b.

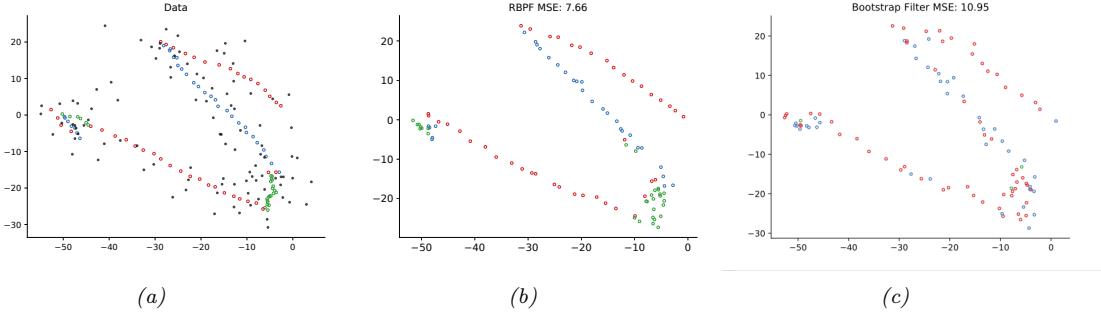


Figure 13.6: Illustration of state estimation for a switching linear model. (a) Black dots are observations, hollow circles are the true location, colors represent the discrete state. (b) Estimate from RBPF. Generated by `rbpf_maneuver.ipynb`. (c) Estimate from bootstrap filter. Generated by `bootstrap_filter_maneuver.ipynb`.

13.4.3 Example: FastSLAM

Consider a robot moving around an environment, such as a maze or indoor office environment. It needs to learn a map of the environment, and keep track of its location (pose) within that map. This problem is known as **simultaneous localization and mapping**, or **SLAM** for short. SLAM is widely used in mobile robotics (see e.g., [SC86; CN01; TBF06] for details). It is also useful in augmented reality, where the task is to recursively estimate the 3d pose of a handheld camera with respect to a set of 2d visual landmarks (this is known as **visual SLAM**, [TUI17; SMT18; Cza+20; DH22]).

Let us assume we can represent the map as the 2d locations of a set of K landmarks, denote them by $\mathbf{l}^1, \dots, \mathbf{l}^K$ (each is a vector in \mathbb{R}^2). (We can use data association to figure out which landmark generated each observation, as discussed in Section 29.9.3.2.) Let \mathbf{r}_t represent the unknown location of the robot at time t . Let $\mathbf{z}_t = (\mathbf{r}_t, \mathbf{l}_t^{1:K})$ be the combined state space. We can then perform online inference so that the robot can update its estimate of its own location, and the landmark locations.

The state transition model is defined as

$$p(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{u}_t) = p(\mathbf{r}_t | \mathbf{r}_{t-1}, \mathbf{l}_{t-1}^{1:K}, \mathbf{u}_t) \prod_{k=1}^K p(\mathbf{l}_t^k | \mathbf{l}_{t-1}^k) \quad (13.68)$$

where $p(\mathbf{r}_t | \mathbf{r}_{t-1}, \mathbf{l}_{t-1}^{1:K}, \mathbf{u}_t)$ specifies how the robot moves given the control signal \mathbf{u}_t and the location of the obstacles $\mathbf{l}_{t-1}^{1:K}$. (Note that in this section, we assume that a human is joysticking the robot through the environment, so $\mathbf{u}_{1:t}$ is given as input, i.e., we do not address the decision-theoretic issue of choosing where to move.)

If the obstacles (landmarks) are static, we can define $p(\mathbf{l}_t^k | \mathbf{l}_{t-1}^k) = \delta(\mathbf{l}_t^k - \mathbf{l}_{t-1}^k)$, which is equivalent to treating the map as an unknown parameter that is shared globally across all time steps. More generally, we can let the landmark locations evolve over time [Mur00].

The observations \mathbf{y}_t measure the distance from \mathbf{r}_t to the set of closest landmarks. Figure 13.8 shows the corresponding graphical model for the case where $K = 2$, and where on the first step it sees landmarks 1 and 2, then just landmark 2, then just landmark 1, etc.

If all the CPDs are linear-Gaussian, then we can use a Kalman filter to maintain our belief state

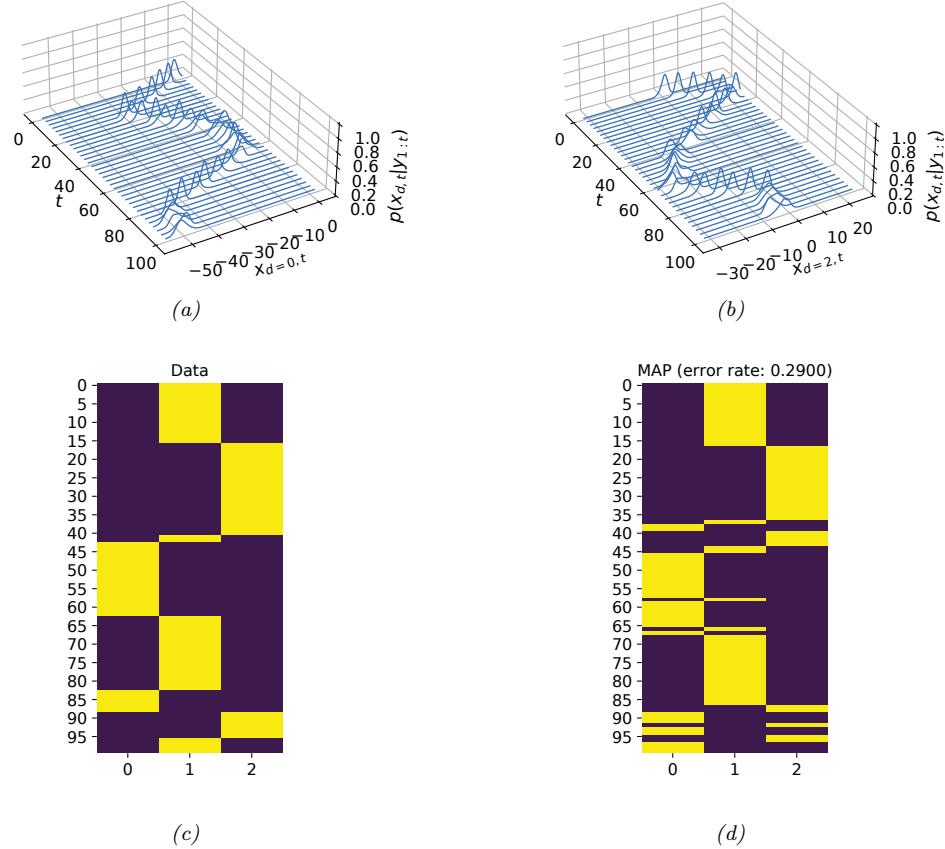


Figure 13.7: Visualizing the posterior from the RBPF algorithm. Top row: Posterior marginals of the location of the object over time, derived from the mixture of Gaussian representation for (a) x location (dimension 0), (b) y location (dimension 2). Bottom row: visualization of the true (c) and predicted (d) discrete states. Generated by [rbpf_maneuver.ipynb](#).

about the location of the robot and the location of the landmarks, $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t})$. In the more general case of a nonlinear model, we can use the EKF (Section 8.3.2) or UKF (Section 8.4.2).

Over time, the uncertainty in the robot's location will increase, due to wheel slippage, etc., but when the robot returns to a familiar location, its uncertainty will decrease again. This is called **closing the loop**, and is illustrated in Figure 13.9(a), where we see the uncertainty ellipses, representing $\text{Cov}[\mathbf{z}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}]$, grow and then shrink.

In addition to visualizing the uncertainty of the robot's location, we can visualize the uncertainty about the map. To do this, consider the posterior precision matrix, $\mathbf{\Lambda}_t = \mathbf{\Sigma}_t^{-1}$. Zeros in the precision

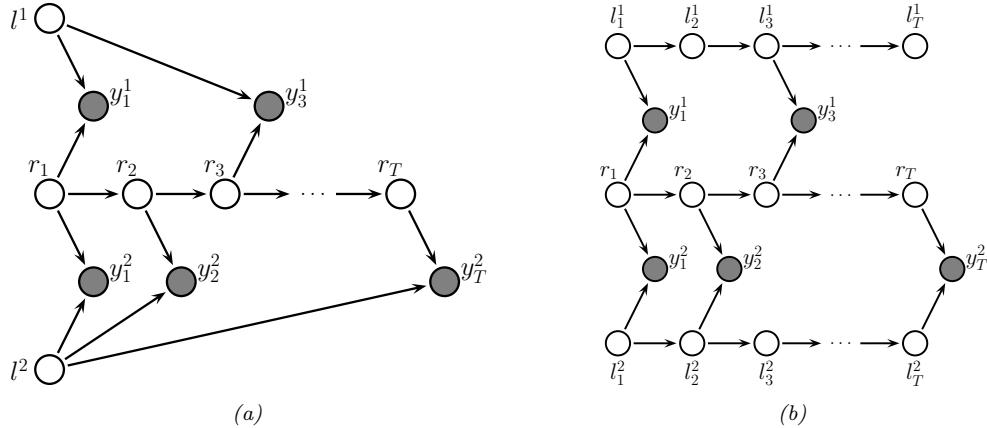


Figure 13.8: Graphical model representing the SLAM problem. l_t^k is the location of landmark k at time t , r_t is the location of the robot at time t , and y_t is the observation vector. In the model on the left, the landmarks are static (so they act like global shared parameters), on the right, their location can change over time. The robot's observations are based on the distance to the nearest landmarks from the current state, denoted $f(r_t, l_t^k)$. The number of observations per time step is variable, depending on how many landmarks are within the range of the sensor. Adapted from Figure 15.A.3 of [KF09a].

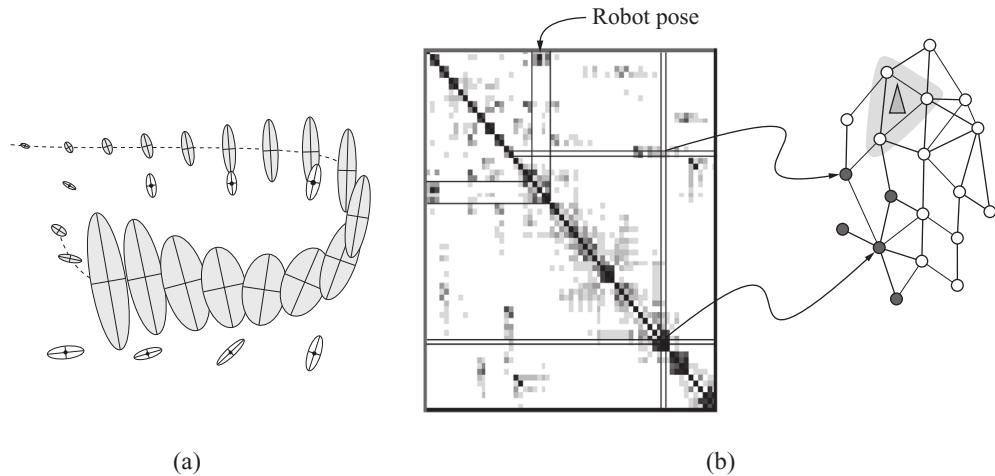


Figure 13.9: Illustration of the SLAM problem. (a) A robot starts at the top left and moves clockwise in a circle back to where it started. We see how the posterior uncertainty about the robot's location increases and then decreases as it returns to a familiar location, closing the loop. If we performed smoothing, this new information would propagate backwards in time to disambiguate the entire trajectory. (b) We show the precision matrix, representing sparse correlations between the landmarks, and between the landmarks and the robot's position (pose). The conditional independencies encoded by the sparse precision matrix can be visualized as a Gaussian graphical model, as shown on the right. From Figure 15.A.3 of [KF09a]. Used with kind permission of Daphne Koller.

matrix correspond to absent edges in the corresponding undirected Gaussian graphical model (GGM, see Section 4.3.5). Initially all the beliefs about landmark locations are uncorrelated (by assumption), so the GGM is a disconnected graph, and Λ_t is diagonal. However, as the robot moves about, it will induce correlation between nearby landmarks. Intuitively this is because the robot is estimating its position based on distance to the landmarks, but the landmarks' locations are being estimated based on the robot's position, so they all become interdependent. This can be seen more clearly from the graphical model in Figure 13.8: it is clear that l^1 and l^2 are not d-separated by $y_{1:t}$, because there is a path between them via the unknown sequence of $r_{1:t}$ nodes. Consequently, the precision matrix becomes denser over time. As a consequence of the precision matrix becoming denser, each inference step takes $O(K^3)$ time. This prevents the method from being applied to large maps.

One way to speed this up is based on the following observation: conditional on knowing the robot's path, $r_{1:t}$, the landmark locations are independent, i.e., $p(l_t | r_{1:t}, y_{1:t}) = \prod_{k=1}^K p(l_t^k | r_{1:t}, y_{1:t})$. This can be seen by looking at the DGM in Figure 13.8. We can therefore sample the trajectory using some proposal, and apply (2d) Kalman filtering to each landmark independently. This is an example of RBPF, and reduces the inference cost to $O(NK)$, where N is the number of particles and K is the number of landmarks.

The overall cost of this technique is $O(NK)$ per step. Fortunately, the number of particles N needed for good performance is quite small, so the algorithm is essentially linear in the number of landmarks, making it quite scalable. This idea was first suggested in [Mur00], who applied it to grid-structured occupancy grids (and used the HMM filter for each particle). It was subsequently extended to landmark-based maps in [Thr+04], using the Kalman filter for each particle; they called the technique **FastSLAM**.

13.5 Extensions of the particle filter

There are many extensions to the basic particle filtering algorithm, such as the following:

- We can increase particle diversity by applying one or more steps of MCMC sampling (Section 12.2) at each PF step using $\pi_t(z_t)$ as the target distribution. This is called the **resample-move** algorithm [DJ11]. It is also possible to use SMC instead of MCMC to diversify the samples [GM17].
- We can extend PF to the case of offline inference; this is called **particle smoothing** (see e.g., [Kla+06]).
- We can extend PF to inference in general graphical models (not just chains) by combining PF with loopy belief propagation (Section 9.4); this is called **non-parametric BP** or **particle BP** (see e.g., [Sud+03; Isa03; Sud+10; Pac+14]).
- We can extend PF to perform inference in static models (e.g., for parameter inference), as we discuss in Section 13.6.

13.6 SMC samplers

In this section, we discuss **SMC samplers** (sequential Monte Carlo samplers), which are a way to apply particle filters to sample from a generic target distribution, $\pi(z) = \tilde{\gamma}(z)/Z$, rather than

requiring the model to be an SSM. Thus SMC is an alternative to MCMC.

The advantages of SMC samplers over MCMC are as follows: we can estimate the normalizing constant Z ; we can more easily develop adaptive versions that tune the transition kernel using the current set of samples; and the method is easier to parallelize (see e.g., [CCS22; Gre+22]).

The method works by defining a sequence of intermediate distributions, $\pi_t(\mathbf{z}_t)$, which we expand to a sequence of distributions over all the past variables, $\bar{\pi}_t(\mathbf{z}_{1:t})$. We then use the particle filtering algorithm to sample from each of these intermediate distributions. By marginalizing all but the final state, we recover samples from the target distribution, $\pi(\mathbf{z}) = \sum_{\mathbf{z}_{1:T-1}} \bar{\pi}_T(\mathbf{z}_{1:T})$, as we explain below. (For more details, see e.g., [Dai+20a; CP20b].)

13.6.1 Ingredients of an SMC sampler

To define an SMC sampler, we need to specify several ingredients:

- A sequence of distributions defined on the same state space, $\pi_t(\mathbf{z}_t) = \tilde{\gamma}_t(\mathbf{z}_t)/Z_t$, for $t = 0 : T$;
- A **forwards kernel** $M_t(\mathbf{z}_t|\mathbf{z}_{t-1})$ (often written as $M_t(\mathbf{z}_{t-1}, \mathbf{z}_t)$), which satisfies $\sum_{\mathbf{z}_t} M_t(\mathbf{z}_t|\mathbf{z}_{t-1}) = 1$. This can be used to propose new samples from our current estimate when we apply particle filtering.
- A **backwards kernel** $L_t(\mathbf{z}_t|\mathbf{z}_{t+1})$ (often written as $L(\mathbf{z}_t, \mathbf{z}_{t+1})$), which satisfies $\sum_{\mathbf{z}_t} L_t(\mathbf{z}_t|\mathbf{z}_{t+1}) = 1$. This allows us to create a sequence of variables by working backwards in time from the final target value to the first time step. In particular, we create the following joint distribution:

$$\bar{\pi}_t(\mathbf{z}_{1:t}) = \pi_t(\mathbf{z}_t) \prod_{s=1}^{t-1} L_s(\mathbf{z}_s|\mathbf{z}_{s+1}) \quad (13.69)$$

This satisfies $\sum_{\mathbf{z}_{1:t-1}} \bar{\pi}_t(\mathbf{z}_{1:t}) = \pi_t(\mathbf{z}_t)$, so if we apply particle filtering to this for $t = 1 : T$, then samples from the “end” of such sequences will be from the target distribution π_t .

With the above ingredients, we can compute the incremental weight at step t using

$$\alpha_t = \frac{\bar{\pi}_t(\mathbf{z}_{1:t})}{\bar{\pi}_{t-1}(\mathbf{z}_{1:t-1})M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \propto \frac{\tilde{\gamma}_t(\mathbf{z}_t)}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \frac{L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t)}{M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.70)$$

This can be plugged into the generic SMC algorithm, Algorithm 13.3.

We still have to specify the forwards and backwards kernels. We will assume the forwards kernel M_t is an MCMC kernel that leaves π_t invariant. We can then define the backwards kernel to be the **time reversal** of the forwards kernel. More precisely, suppose we define L_{t-1} so it satisfies

$$\pi_t(\mathbf{z}_t)L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t) = \pi_t(\mathbf{z}_{t-1})M_t(\mathbf{z}_t|\mathbf{z}_{t-1}) \quad (13.71)$$

In this case, the incremental weight simplifies as follows:

$$\alpha_t = \frac{Z_t \pi_t(\mathbf{z}_t) L_{t-1}(\mathbf{z}_{t-1}|\mathbf{z}_t)}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.72)$$

$$= \frac{Z_t \pi_t(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})}{Z_{t-1} \pi_{t-1}(\mathbf{z}_{t-1}) M_t(\mathbf{z}_t|\mathbf{z}_{t-1})} \quad (13.73)$$

$$= \frac{\tilde{\gamma}_t(\mathbf{z}_{t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} \quad (13.74)$$

We can use any kind of MCMC kernel for M_t . For example, if the parameters are real valued and unconstrained, we can use a Markov kernel that corresponds to K steps of a random walk Metropolis-Hastings sampler. We can set the covariance of the proposal to $\delta^2 \hat{\Sigma}_{t-1}$, where $\hat{\Sigma}_{t-1}$ is the empirical covariance of the weighted samples from the previous step, $(W_{t-1}^{1:N}, \mathbf{z}_{t-1}^{1:N})$, and $\delta = 2.38D^{-3/2}$ (which is the optimal scaling parameter for RWMH). In high dimensional problems, we can use gradient based Markov kernels, such as HMC [BCJ20] and NUTS [Dev+21]. For binary state spaces, we can use the method of [SC13].

13.6.2 Likelihood tempering (geometric path)

There are many ways to specify the intermediate target distributions. In the **geometric path** method, we specify the intermediate distributions to be

$$\tilde{\gamma}_t(\mathbf{z}) = \tilde{\gamma}_0(\mathbf{z})^{1-\lambda_t} \tilde{\gamma}(\mathbf{z})^{\lambda_t} \quad (13.75)$$

where $0 = \lambda_0 < \lambda_1 < \dots < \lambda_T = 1$ are **inverse temperature** parameters, and $\tilde{\gamma}_0$ is the initial proposal. If we apply particle filtering to this model, but “turn off” the resampling step, the method becomes equivalent to **annealed importance sampling** (Section 11.5.4).

In the context of Bayesian parameter inference, we often denote the latent variable \mathbf{z} by $\boldsymbol{\theta}$, we define $\tilde{\gamma}_0(\boldsymbol{\theta}) \propto \pi_0(\boldsymbol{\theta})$ as the prior, and $\tilde{\gamma}(\boldsymbol{\theta}) = \pi_0(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$ as the posterior. We can then define the intermediate distributions to be

$$\tilde{\gamma}_t(\boldsymbol{\theta}) = \pi_0(\boldsymbol{\theta})^{1-\lambda_t} \pi_0(\boldsymbol{\theta})^{\lambda_t} p(\mathcal{D}|\boldsymbol{\theta})^{\lambda_t} = \pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.76)$$

where $\mathcal{E}(\boldsymbol{\theta}) = -\log p(\mathcal{D}, \boldsymbol{\theta})$ is the energy (potential) function. The incremental weights are given by

$$\alpha_t(\boldsymbol{\theta}) = \frac{\pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_t \mathcal{E}(\boldsymbol{\theta})]}{\pi_0(\boldsymbol{\theta})^{1-\lambda_t} \exp[-\lambda_{t-1} \mathcal{E}(\boldsymbol{\theta})]} = \exp[-\delta_t \mathcal{E}(\boldsymbol{\theta})] \quad (13.77)$$

where $\lambda_t = \lambda_{t-1} + \delta_t$.

For this method to work well, it is important to choose the λ_t so that the successive distributions are “equidistant”; this is called **adaptive tempering**. In the case of a Gaussian prior and Gaussian energy, one can show [CP20b] that this can be achieved by picking $\lambda_t = (1 + \gamma)^{t+1} - 1$, where $\gamma > 0$ is some constant. Thus we should increase λ slowly at first, and then make bigger and bigger steps.

In practice we can estimate λ_t by setting $\lambda_t = \lambda_{t-1} + \delta_t$, where

$$\delta_t = \underset{\delta \in [0, 1 - \lambda_{t-1}]}{\operatorname{argmin}} (\operatorname{ESSLW}(\{-\delta \mathcal{E}(\boldsymbol{\theta}_t^n)\}) - \operatorname{ESS}_{\min}) \quad (13.78)$$

where $\operatorname{ESSLW}(\{l_n\}) = \operatorname{ESS}(\{e^{l_n}\})$ computes the ESS (Equation (13.37)) from the log weights, $l_n = \log \tilde{w}^n$. This ensures the change in the ESS across steps is close to the desired minimum ESS, typically $0.5N$. (If there is no solution for δ in the interval, we set $\delta_t = 1 - \lambda_{t-1}$.) See Algorithm 13.5 for the overall algorithm.

13.6.2.1 Example: sampling from a 1d bimodal distribution

Consider the simple distribution

$$p(\boldsymbol{\theta}) \propto \mathcal{N}(\boldsymbol{\theta} | \mathbf{0}, \mathbf{I}) \exp(-\mathcal{E}(\boldsymbol{\theta})) \quad (13.79)$$

Algorithm 13.5: SMC with adaptive tempering

```

1  $\lambda_{-1} = 0, t = -1, W_{-1}^n = 1$ 
2 while  $\lambda_t < 1$  do
3    $t = t + 1$ 
4   if  $t = 0$  then
5      $\theta_0^n \sim \pi_0(\theta)$ 
6   else
7      $A_t^{1:N} = \text{Resample}(W_{t-1}^{1:N})$ 
8      $\theta_t^n \sim M_{\lambda_{t-1}}(\theta_{t-1}^{A_t^n}, \cdot)$ 
9   Compute  $\delta_t$  using Equation (13.78)
10   $\lambda_t = \lambda_{t-1} + \delta_t$ 
11   $\tilde{w}_t^n = \exp[-\delta\mathcal{E}(\theta_t^n)]$ 
12   $W_t^n = \tilde{w}_t^n / (\sum_{m=1}^N \tilde{w}_t^m)$ 

```

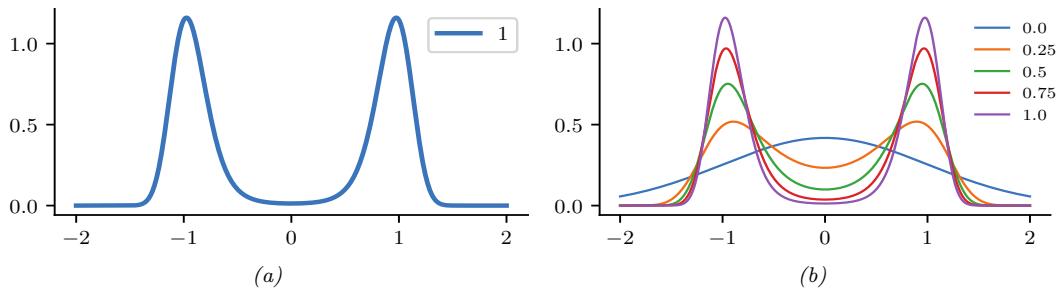


Figure 13.10: (a) Illustration of a bimodal target distribution. (b) Tempered versions of the target at different inverse temperatures, from $\lambda_T = 1$ down to $\lambda_1 = 0$. Generated by `smc_tempered_1d_bimodal.ipynb`.

where $\mathcal{E}(\boldsymbol{\theta}) = c(||\boldsymbol{\theta}||^2 - 1)^2$. We plot this in 1d in Figure 13.10a for $c = 5$; we see that it has a bimodal shape, since the low energy states correspond to parameter vectors whose norm is close to 1.

SMC is particularly useful for sampling from multimodal distributions, which can be provably hard to efficiently sample from using other methods, including HMC [MPS18], since gradients only provide local information about the curvature. As an example, in Figure 13.11a and Figure 13.11b we show the result of applying HMC (Section 12.5) and NUTS (Section 12.5.4.1) to this problem. We see that both algorithms get stuck near the initial state of $\theta_0 = 1$.

In Figure 13.10b, we show tempered versions of the target distribution at 5 different temperatures, chosen uniformly in the interval $[0, 1]$. We see that at $\lambda_1 = 0$, the tempered target is equal to the Gaussian prior (blue line), which is easy to sample from. Each subsequent distribution is close to the previous one, so SMC can track the change until it ends up at the target distribution with $\lambda_T = 1$, as shown in Figure 13.11c.

These SMC results were obtained using the adaptive tempering scheme described above. In Figure 13.11d we see that initially the temperature is small, and then it increases exponentially. The algorithm takes 8 steps until $\lambda_T \geq 1$.

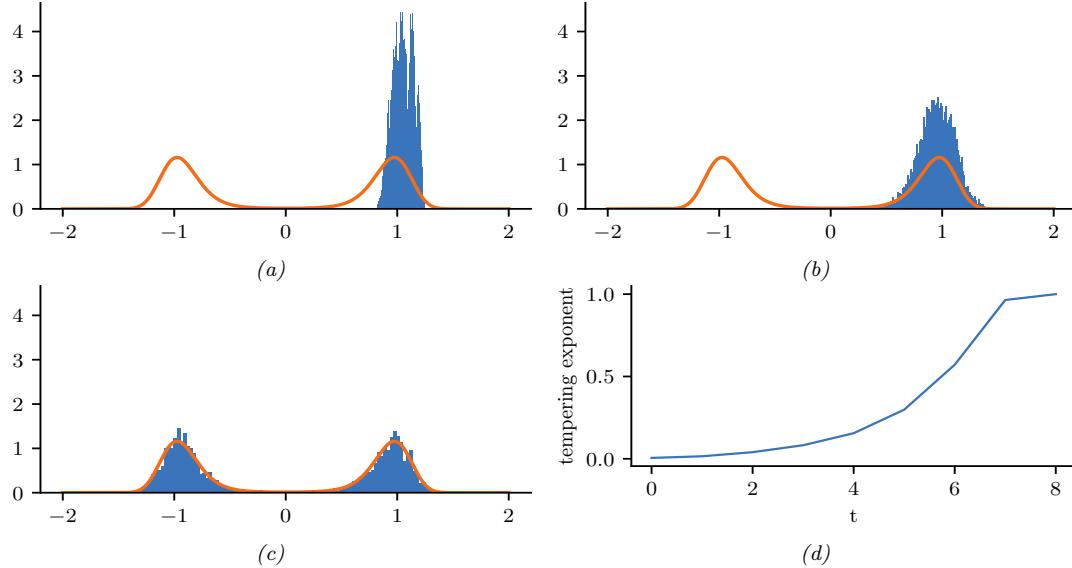


Figure 13.11: Sampling from the bimodal distribution in Figure 13.10a. (a) HMC. (b) NUTS. (c) Tempered SMC with HMC kernel (single step). (d) Adaptive inverse temperature schedule. Generated by `smc_tempered_1d_bimodal.ipynb`.

13.6.3 Data tempering

If we have a set of iid observations, we can define the t 'th target to be

$$\tilde{\gamma}_t(\boldsymbol{\theta}) = p(\boldsymbol{\theta})p(\mathbf{y}_{1:t}|\boldsymbol{\theta}) \quad (13.80)$$

We can now apply SMC to this model. From Equation (13.74), the incremental weight becomes

$$\alpha_t(\boldsymbol{\theta}) = \frac{\tilde{\gamma}_t(\mathbf{z}_{t-1})}{\tilde{\gamma}_{t-1}(\mathbf{z}_{t-1})} = \frac{p(\boldsymbol{\theta})p(\mathbf{y}_{1:t}|\boldsymbol{\theta})}{p(\boldsymbol{\theta})p(\mathbf{y}_{1:t-1}|\boldsymbol{\theta})} = p(\mathbf{y}_t|\mathbf{y}_{1:t-1}, \boldsymbol{\theta}) \quad (13.81)$$

This can be plugged into the generic SMC algorithm in Algorithm 13.3.

Unfortunately, to sample from the MCMC kernel will typically take $O(t)$ time, since the MH accept/reject step requires computing $p(\boldsymbol{\theta}') \prod_{i=1}^t p(\mathbf{y}_{1i}|\boldsymbol{\theta}')$ for any proposed $\boldsymbol{\theta}'$. Hence the total cost is $O(T^2)$ if there are T observations. To reduce this, we can only sample parameters at times t when the ESS drops below a certain level; in the remaining steps, we just grow the sequence deterministically by repeating the previously sampled value. This technique was proposed in [Cho02], who called it the **iterated batch importance sampling** or **IBIS** algorithm.

13.6.3.1 Example: IBIS for a 1d Gaussian

In this section, we give a simple example of IBIS applied to data from a 1d Gaussian, $y_t \sim \mathcal{N}(\mu = 3.14, \sigma = 1)$ for $t = 1 : 30$. The unknowns are $\boldsymbol{\theta} = (\mu, \sigma)$. The prior is $p(\boldsymbol{\theta}) = \mathcal{N}(\mu|0, 1)\text{Ga}(\sigma|a =$

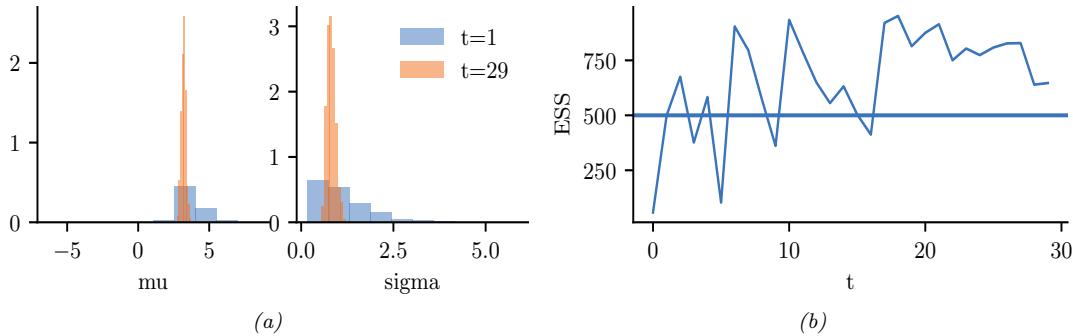


Figure 13.12: Illustration of IBIS applied to 30 samples from $\mathcal{N}(\mu = 3.14, \sigma = 1)$. (a) Posterior approximation after $t = 1$ and $t = 29$ observations. (b) Effective sample size over time. The sudden jumps up occur whenever resampling is triggered, which happens when the ESS drops below 500. Generated by `smc_ibis_1d.ipynb`.

$1, b = 1$). We use IBIS with an adaptive RWMH kernel. We use $N = 20$ particles, each updated for $K = 50$ MCMC steps, so we collect 1000 samples per time step.

Figure 13.12a shows the approximate posterior after $t = 1$ and $t = 29$ time steps. We see that the posterior concentrates on the true values of $\mu = 3.14$ and $\sigma = 1$.

Figure 13.12b plots the ESS vs time. The number of particles is 1000, and resampling (and MCMC moves) is triggered whenever this drops below 500. We see that we only need to invoke MCMC updates 3 times.

13.6.4 Sampling rare events and extrema

Suppose we want to sample values from $\pi_0(\boldsymbol{\theta})$ conditioned on the event that $S(\boldsymbol{\theta}) > \lambda^*$, where S is some score or “fitness” function. If λ^* is in the tail of the score distribution, this corresponds to sampling a **rare event**, which can be hard.

One approach is to use SMC to sample from a sequence of distributions with gradually increasing thresholds:

$$\pi_t(\boldsymbol{\theta}) = \frac{1}{Z_t} \mathbb{I}(S(\boldsymbol{\theta}) \geq \lambda_t) \pi_0(\boldsymbol{\theta}) \quad (13.82)$$

with $\lambda_0 < \dots < \lambda_T = \lambda^*$. We can then use likelihood tempering, where the “likelihood” is the function

$$G_t(\boldsymbol{\theta}_t) = \mathbb{I}(S(\boldsymbol{\theta}_t) \geq \lambda_t) \quad (13.83)$$

We can use SMC to generate samples from the final distribution π_T . We may also be interested in estimating

$$Z_T = p(S(\boldsymbol{\theta}) \geq \lambda_T) \quad (13.84)$$

where the probability is taken wrt $\pi_0(\boldsymbol{\theta})$.

We can adaptively set the thresholds λ_t as follows: at each step, sort the samples by their score, and set λ_t to the α 'th highest quantile. For example, if we set $\alpha = 0.5$, we keep the top 50% fittest particles. This ensures the ESS equals the minimum threshold at each step. For details, see [Cér+12].

Note that this method is very similar to the **cross-entropy method** (Section 6.7.5). The difference is that CEM fits a parametric distribution (e.g., a Gaussian) to the particles at each step and samples from that, rather than using a Markov kernel.

13.6.5 SMC-ABC and likelihood-free inference

The term **likelihood-free inference** refers to estimating the parameters $\boldsymbol{\theta}$ of a blackbox from which we can sample data, $\mathbf{y} \sim p(\cdot|\boldsymbol{\theta})$, but where we cannot evaluate $p(\mathbf{y}|\boldsymbol{\theta})$ pointwise. Such models are called simulators, so this approach to inference is also called **simulation-based inference** (see e.g., [Nea+08; CBL20; Gou+96]). These models are also called **implicit models** (see Section 26.1).

If we want to approximate the posterior of a model with no known likelihood, we can use **approximate Bayesian computation** or **ABC** (see e.g., [Bea19; SFB18; Gut+14; Pes+21]). In this setting, we sample both parameters $\boldsymbol{\theta}$ and synthetic data \mathbf{y} such that the synthetic data (generated from $\boldsymbol{\theta}$) is sufficiently close to the observed data \mathbf{y}^* , as judged by some distance score, $d(\mathbf{y}, \mathbf{y}^*) < \epsilon$. (For high dimensional problems, we typically require $d(\mathbf{s}(\mathbf{y}), \mathbf{s}(\mathbf{y}^*)) < \epsilon$, where $\mathbf{s}(\mathbf{y})$ is a low-dimensional summary statistic of the data.)

In **SMC-ABC**, we gradually decrease the discrepancy ϵ to get a series of distributions as follows:

$$\pi_t(\boldsymbol{\theta}, \mathbf{y}) = \frac{1}{Z_t} \pi_0(\boldsymbol{\theta}) p(\mathbf{y}|\boldsymbol{\theta}) \mathbb{I}(d(\mathbf{y}, \mathbf{y}^*) < \epsilon_t) \quad (13.85)$$

where $\epsilon_0 > \epsilon_1 > \dots$. This is similar to the rare event SMC samplers in Section 13.6.4, except that we can't directly evaluate the quality of a candidate, $\boldsymbol{\theta}$. Instead we must first convert it to data space and make the comparison there. For details, see [DMDJ12].

Although SMC-ABC is popular in some fields, such as genetics and epidemiology, this method is quite slow and does not scale to high dimensional problems. In such settings, a more efficient approach is to train a generative model to **emulate** the simulator; if this model is parametric with a tractable likelihood (e.g., a flow model), we can use the usual methods for posterior inference of its parameters (including gradient based methods like HMC). See e.g., [Bre+20a] for details.

13.6.6 SMC²

We have seen how SMC can be a useful alternative to MCMC. However it requires that we can efficiently evaluate the likelihood ratio terms $\frac{\gamma_t(\boldsymbol{\theta}_t)}{\gamma_{t-1}(\boldsymbol{\theta}_t)}$. In cases where this is not possible (e.g., for latent variable models), we can use SMC (specifically the estimate \hat{Z}_t in Equation (13.10)) as a subroutine to approximate these likelihoods. This is called **SMC²**. For details, see [CP20b, Ch. 18].

13.6.7 Variational filtering SMC

One way to improve SMC is to learn a proposal distribution (e.g., using a neural network) such that the approximate posterior, $\hat{\pi}_T(\mathbf{z}_{1:T}; \boldsymbol{\phi}, \boldsymbol{\theta})$, is close to the target posterior, $\pi_T(\mathbf{z}_{1:T}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the model parameters, and $\boldsymbol{\phi}$ are the proposal parameters (which may depend on $\boldsymbol{\theta}$). One can show

[Nae+18] that the KL divergence between these distributions can be bounded as follows:

$$0 \leq D_{\text{KL}}(\mathbb{E}[\hat{\pi}_T(\mathbf{z}_{1:T})] \parallel \pi_T(\mathbf{z}_{1:T})) \leq -\mathbb{E}\left[\log \frac{\hat{Z}_T}{Z_T}\right] \quad (13.86)$$

where

$$Z_T(\boldsymbol{\theta}) = p_{\boldsymbol{\theta}}(\mathbf{y}_{1:T}) = \int p_{\boldsymbol{\theta}}(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) d\mathbf{z}_{1:T} \quad (13.87)$$

Hence

$$\mathbb{E}\left[\log \hat{Z}_T(\boldsymbol{\theta}, \boldsymbol{\phi})\right] \leq \mathbb{E}[\log Z_T(\boldsymbol{\theta})] = \log Z_T(\boldsymbol{\theta}) \quad (13.88)$$

Thus we can use SMC sampling to compute an unbiased approximation to $\mathbb{E}[\log \hat{Z}_T(\boldsymbol{\theta}, \boldsymbol{\phi})]$, which is a lower bound on the evidence (log marginal likelihood).

We can now maximize this lower bound wrt $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ using SGD, as a way to learn both proposals and the model. Unfortunately, computing the gradient of the bound is tricky, since the resampling step is non-differentiable. However, in practice one can ignore the dependence of the resampling operator on the parameters, or one can use differentiable approximations (see e.g., [Ros+22]). This overall approach was independently proposed in several papers: the **FIVO** (filtering variational objective) paper [Mad+17], the **variational SMC** paper [Nae+18] and the **auto-encoding SMC** paper [Le+18].

13.6.8 Variational smoothing SMC

The methods in Section 13.6.7 use SMC in which the target distributions are defined to be the filtered distributions, $\pi_t(\mathbf{z}_{1:t}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t} | \mathbf{y}_{1:t})$; this is called **filtering SMC**. Unfortunately, this can work poorly when fitting models to offline sequence data, since at time t , all future observations are ignored in the objective, no matter how good the proposal. This can create situations where future observations are unlikely given the current set of sampled trajectories, which can result in particle impoverishment and high variance in the estimate of the lower bound.

Recently, a new method called **SIXO** (smoothing inference with twisted objectives) was proposed in [Law+22] that uses the smoothing distributions as targets, $\pi_t(\mathbf{z}_{1:t}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t} | \mathbf{y}_{1:T})$, to create a much lower variance variational lower bound. Of course it is impossible to directly compute this posterior, but we can approximate it using **twisted particle filters** [WL14a; AL+16]. In this approach, we approximate the (unnormalized) posterior using

$$p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:T}) = p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_{1:t}, \mathbf{y}_{1:t}) \quad (13.89)$$

$$= p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t) \quad (13.90)$$

$$\approx p_{\boldsymbol{\theta}}(\mathbf{z}_{1:t}, \mathbf{y}_{1:t}) r_{\boldsymbol{\psi}}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \quad (13.91)$$

where $r_{\boldsymbol{\psi}}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \approx p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t)$ is the **twisting function**, which acts as a “**lookahead function**”.

One way to approximate the twisting function is to note that

$$p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T} | \mathbf{z}_t) = \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t | \mathbf{y}_{t+1:T}) p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)} \propto \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t | \mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)} \quad (13.92)$$

where we drop terms that are independent of \mathbf{z}_t since such terms will cancel out when we normalize the sampling weights. We can approximate the density ratio using the binary classifier method of Section 2.7.5. To do this, we define one distribution to be $p_1 = p_{\boldsymbol{\theta}}(\mathbf{z}_t, \mathbf{y}_{t+1:T})$ and the other to be $p_2 = p_{\boldsymbol{\theta}}(\mathbf{z}_t)p_{\boldsymbol{\theta}}(\mathbf{y}_{t+1:T})$, so that $p_1/p_2 = \frac{p_{\boldsymbol{\theta}}(\mathbf{z}_t|\mathbf{y}_{t+1:T})}{p_{\boldsymbol{\theta}}(\mathbf{z}_t)}$. We can easily draw a sample $(\mathbf{z}_{1:T}, \mathbf{y}_{1:T}) \sim p_{\boldsymbol{\theta}}$ using ancestral sampling, from which we can compute $(\mathbf{z}_t, \mathbf{y}_{t+1:T}) \sim p_1$ by marginalization. We can also sample a fresh sequence from $(\tilde{\mathbf{z}}_{1:T}, \tilde{\mathbf{y}}_{1:T}) \sim p_{\boldsymbol{\theta}}$ from which we can compute $(\tilde{\mathbf{z}}_t, \tilde{\mathbf{y}}_{t+1:T}) \sim p_2$ by marginalization. We then use $(\mathbf{z}_t, \mathbf{y}_{t+1:T})$ as a positive example and $(\tilde{\mathbf{z}}_t, \tilde{\mathbf{y}}_{t+1:T})$ as a negative example when training the binary classifier, $r_{\boldsymbol{\psi}}(\mathbf{y}_{t+1:T}, \mathbf{z}_t)$.

Once we have updated the twisting parameters $\boldsymbol{\psi}$, we can rerun SMC to get a tighter lower bound on the log marginal likelihood, which we can then optimize wrt the model parameters $\boldsymbol{\theta}$ and proposal parameters $\boldsymbol{\phi}$. Thus the overall method is a stochastic variational EM-like method for optimziing the bound

$$\mathcal{L}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \triangleq \mathbb{E} \left[\log \hat{Z}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \right] \quad (13.93)$$

$$\leq \log \mathbb{E} \left[\hat{Z}_{\text{SIXO}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\psi}, \mathbf{y}_{1:T}) \right] = \log p_{\boldsymbol{\theta}}(\mathbf{y}_{1:T}) \quad (13.94)$$

In [Law+22] they prove the following: suppose the true model p^* is an SSM in which the optimal proposal function for the model satisfies $p^*(\mathbf{z}_t|\mathbf{z}_{1:t-1}, \mathbf{y}_{1:T}) \in \mathcal{Q}$, and the optimal lookahead function for the model satisfies $p^*(\mathbf{y}_{t+1:T}|\mathbf{z}_t) \in \mathcal{R}$. Furthermore, assume the SIXO objective has a unique maximizer. Then, at the optimum, we have that the learned proposal $q_{\boldsymbol{\phi}^*}(\mathbf{z}_t|\mathbf{z}_{1:t-1}, \mathbf{y}_{1:T}) \in \mathcal{Q}$ is equal to the optimal proposal, the learned twisting function $r_{\boldsymbol{\psi}^*}(\mathbf{y}_{t+1:T}, \mathbf{z}_t) \in \mathcal{R}$ is equal to the optimal lookahead, and the lower bound is tight (i.e., $\mathcal{L}_{\text{SIXO}}(\boldsymbol{\theta}^*, \boldsymbol{\phi}^*, \boldsymbol{\psi}^*) = p^*(\mathbf{y}_{1:T})$) for any number of samples $N_s \geq 1$ and for any kind of SSM p^* . (This is in contrast to the FIVO bound, whiere the bound does not usually become tight.)

PART III

Prediction

14 Predictive models: an overview

14.1 Introduction

The vast majority of machine learning is concerned with tackling a single problem, namely learning to predict outputs \mathbf{y} from inputs \mathbf{x} using some function f that is estimated from a labeled training set $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n) : n = 1 : N\}$, for $\mathbf{x}_n \in \mathcal{X} \subseteq \mathbb{R}^D$ and $\mathbf{y}_n \in \mathcal{Y} \subseteq \mathbb{R}^C$. We can model our uncertainty about the correct output for a given input using a conditional probability model of the form $p(\mathbf{y}|f(\mathbf{x}))$. When \mathcal{Y} is a discrete set of labels, this is called (in the ML literature) a **discriminative model**, since it lets us discriminate (distinguish) between the different possible values of \mathbf{y} . If the output is real-valued, $\mathcal{Y} = \mathbb{R}$, this is called a **regression model**. (In the statistics literature, the term “regression model” is used in both cases, even if \mathcal{Y} is a discrete set.) We will use the more generic term **“predictive model”** to refer to such models.

A predictive model can be considered as a special case of a conditional generative model (discussed in Chapter 20). In a predictive model, the output is usually low dimensional, and there is a single best answer that we want to predict. However, in most generative models, the output is usually high dimensional, such as images or sentences, and there may be many correct outputs for any given input. We will discuss a variety of types of predictive model in Section 14.1.1, but we defer the details to subsequent chapters. The rest of this chapter then discusses issues that are relevant to all types of predictive model, regardless of the specific form, such as evaluation.

14.1.1 Types of model

There are many different kinds of predictive model $p(\mathbf{y}|\mathbf{x})$. The biggest distinction is between **parametric models**, that have a fixed number of parameters independent of the size of the training set, and **non-parametric models** that have a variable number of parameters that grows with the size of the training set. Non-parametric models are usually more flexible, but can be slower to use for prediction. Parametric models are usually less flexible, but are faster to use for prediction.

Most non-parametric models are based on comparing a test input \mathbf{x} to some or all of the stored training examples $\{\mathbf{x}_n, n = 1 : N\}$, using some form of similarity, $s_n = \mathcal{K}(\mathbf{x}, \mathbf{x}_n) \geq 0$, and then predicting the output using some weighted combination of the training labels, such as $\hat{\mathbf{y}} = \sum_{n=1}^N s_n \mathbf{y}_n$. A typical example is a Gaussian process, which we discuss in Chapter 18. Other examples, such as K -nearest neighbor models, are discussed in the prequel to this book, [Mur22].

Most parametric models have the form $p(\mathbf{y}|\mathbf{x}) = p(\mathbf{y}|f(\mathbf{x}; \boldsymbol{\theta}))$, where f is some kind of function that predicts the parameters (e.g., the mean, or logits) of the output distribution (e.g., Gaussian or categorical). There are many kinds of function we can use. If f is a linear function of $\boldsymbol{\theta}$ (i.e.,

$f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \phi(\mathbf{x})$ for some *fixed* feature transformation ϕ), then the model is called a generalized linear model or GLM, which we discuss in Chapter 15. If f is a non-linear, but differentiable, function of $\boldsymbol{\theta}$ (e.g., $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\theta}_2^\top \phi(\mathbf{x}; \boldsymbol{\theta}_1)$ for some learnable function $\phi(\mathbf{x}; \boldsymbol{\theta}_1)$), then it is common to represent f using a neural network (Chapter 16). Other types of predictive model, such as decision trees and random forests, are discussed in the prequel to this book, [Mur22].

14.1.2 Model fitting using ERM, MLE, and MAP

In this section, we briefly discuss some methods used for fitting (parametric) models. The most common approach is to use **maximum likelihood estimation** or **MLE**, which amounts to solving the following optimization problem:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} p(\mathcal{D}|\boldsymbol{\theta}) = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} \log p(\mathcal{D}|\boldsymbol{\theta}) \quad (14.1)$$

If the dataset is N iid data samples, the likelihood decomposes into a product of terms, $p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^N p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$. Thus we can instead minimize the following (scaled) **negative log likelihood**:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^N [-\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})] \quad (14.2)$$

We can generalize this by replacing the **log loss** $\ell_n(\boldsymbol{\theta}) = -\log p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$ with a more general loss function to get

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} r(\boldsymbol{\theta}) \quad (14.3)$$

where $r(\boldsymbol{\theta})$ is the **empirical risk**

$$r(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \ell_n(\boldsymbol{\theta}) \quad (14.4)$$

This approach is called **empirical risk minimization** or **ERM**.

ERM can easily result in **overfitting**, so it is common to add a penalty or regularizer term to get

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} r(\boldsymbol{\theta}) + \lambda C(\boldsymbol{\theta}) \quad (14.5)$$

where $\lambda \geq 0$ controls the degree of regularization, and $C(\boldsymbol{\theta})$ is some complexity measure. If we use log loss, and we define $C(\boldsymbol{\theta}) = -\log \pi_0(\boldsymbol{\theta})$, where $\pi_0(\boldsymbol{\theta})$ is some prior distribution, and we use $\lambda = 1$, we recover the **MAP estimate**

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmax}} \log p(\mathcal{D}|\boldsymbol{\theta}) + \log \pi_0(\boldsymbol{\theta}) \quad (14.6)$$

This can be solved using standard optimization methods (see Chapter 6).

14.1.3 Model fitting using Bayes, VI, and generalized Bayes

Another way to prevent overfitting is to estimate a *probability distribution over parameters*, $q(\boldsymbol{\theta})$, instead of a point estimate. That is, we can try to estimate the ERM in expectation:

$$\hat{q} = \underset{q \in \mathcal{P}(\Theta)}{\operatorname{argmin}} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] \quad (14.7)$$

If $\mathcal{P}(\Theta)$ is the space of all probability distributions over parameters, then the solution will converge to a delta function that puts all its probability on the MLE. Thus this approach, on its own, will not prevent overfitting. However, we can regularize the problem by preventing the distribution from moving too far from the prior. If we measure the divergence between q and the prior using KL divergence, we get

$$\hat{q} = \underset{q \in \mathcal{P}(\Theta)}{\operatorname{argmin}} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] + \frac{1}{\lambda} D_{\text{KL}}(q \parallel \pi_0) \quad (14.8)$$

The solution to this problem is known as the **Gibbs posterior**, and is given by the following:

$$\hat{q}(\boldsymbol{\theta}) = \frac{e^{-\lambda r(\boldsymbol{\theta})} \pi_0(\boldsymbol{\theta})}{\int e^{-\lambda r(\boldsymbol{\theta}')} \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.9)$$

This is widely used in the **PAC-Bayes** community (see e.g., [Alq21]).

Now suppose we use log loss, and set $\lambda = N$, to get

$$\hat{q}(\boldsymbol{\theta}) = \frac{e^{\sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta})} \pi_0(\boldsymbol{\theta})}{\int e^{\sum_{n=1}^N \log p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}')} \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.10)$$

Then the resulting distribution is equivalent to the Bayes posterior:

$$\hat{q}(\boldsymbol{\theta}) = \frac{p(\mathcal{D} | \boldsymbol{\theta}) \pi_0(\boldsymbol{\theta})}{\int p(\mathcal{D} | \boldsymbol{\theta}') \pi_0(\boldsymbol{\theta}') d\boldsymbol{\theta}'} \quad (14.11)$$

Often computing the Bayes posterior is intractable. We can simplify the problem by restricting attention to a limited family of distributions, $\mathcal{Q}(\Theta) \subset \mathcal{P}(\Theta)$. This gives rise to the following objective:

$$\hat{q} = \underset{q \in \mathcal{Q}(\Theta)}{\operatorname{argmin}} \mathbb{E}_{q(\boldsymbol{\theta})} [-\log p(\mathcal{D} | \boldsymbol{\theta})] + D_{\text{KL}}(q \parallel \pi_0) \quad (14.12)$$

This is known as **variational inference**; see Chapter 10 for details.

We can generalize this by replacing the negative log likelihood with a general risk, $r(\boldsymbol{\theta})$. Furthermore, we can replace the KL with a general divergence, $D(q || \pi_0)$, which we can weight using a general λ . This gives rise to the following objective:

$$\hat{q} = \underset{q \in \mathcal{Q}(\Theta)}{\operatorname{argmin}} \mathbb{E}_{q(\boldsymbol{\theta})} [r(\boldsymbol{\theta})] + \lambda D(q || \pi_0) \quad (14.13)$$

This is called **generalized Bayesian inference** [BHW16; KJD19; KJD21].

14.2 Evaluating predictive models

In this section we discuss how to evaluate the quality of a trained discriminative model.

14.2.1 Proper scoring rules

It is common to measure performance of a predictive model using a **proper scoring rule** [GR07], which is defined as follows. Let $S(p_{\theta}, (y, \mathbf{x}))$ be the score for predictive distribution $p_{\theta}(y|\mathbf{x})$ when given an event $y|\mathbf{x} \sim p^*(y|\mathbf{x})$, where p^* is the true conditional distribution. (If we want to evaluate a Bayesian model, where we marginalize out θ rather than condition on it, we just replace $p_{\theta}(y|\mathbf{x})$ with $p(y|\mathbf{x}) = \int p_{\theta}(y|\mathbf{x})p(\theta|\mathcal{D})d\theta$.) The expected score is defined by

$$S(p_{\theta}, p^*) = \int p^*(\mathbf{x})p^*(y|\mathbf{x})S(p_{\theta}, (y, \mathbf{x}))dyd\mathbf{x} \quad (14.14)$$

A proper scoring rule is one where $S(p_{\theta}, p^*) \leq S(p^*, p^*)$, with equality iff $p_{\theta}(y|\mathbf{x}) = p^*(y|\mathbf{x})$. Thus maximizing such a proper scoring rule will force the model to match the true probabilities.

The log-likelihood, $S(p_{\theta}, (y, \mathbf{x})) = \log p_{\theta}(y|\mathbf{x})$, is a proper scoring rule. This follows from Gibbs inequality:

$$S(p_{\theta}, p^*) = \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p_{\theta}(y|\mathbf{x})] \leq \mathbb{E}_{p^*(\mathbf{x})p^*(y|\mathbf{x})} [\log p^*(y|\mathbf{x})] \quad (14.15)$$

Therefore minimizing the NLL (aka log loss) should result in well-calibrated probabilities. However, in practice, log-loss can over-emphasize tail probabilities [QC+06].

A common alternative is to use the **Brier score** [Bri50], which is defined as follows:

$$S(p_{\theta}, (y, \mathbf{x})) \triangleq \frac{-1}{C} \sum_{c=1}^C (p_{\theta}(y=c|\mathbf{x}) - \mathbb{I}(y=c))^2 \quad (14.16)$$

This is proportional to the squared error of the predictive distribution $\mathbf{p} = p(1:C|\mathbf{x})$ compared to the one-hot label distribution \mathbf{y} . (We add a negative sign to the original definition so that larger values (less negative) are better, to be consistent with the conventions above.) Since it is based on squared error, the Brier score is less sensitive to extremely rare or extremely common classes. The Brier score is also a proper scoring rule.

14.2.2 Calibration

A model whose predicted probabilities match the empirical frequencies is said to be **calibrated** [Daw82; NMC05; Guo+17]. For example, if a classifier predicts $p(y=c|\mathbf{x}) = 0.9$, then we expect this to be the true label about 90% of the time. A well-calibrated model is useful to avoid making the wrong decision when the outcome is too uncertain. In the sections below, we discuss some ways to measure and improve calibration.

14.2.2.1 Expected calibration error

To assess calibration, we divide the predicted probabilities into a finite set of bins or buckets, and then assess the discrepancy between the empirical probability and the predicted probability by counting.

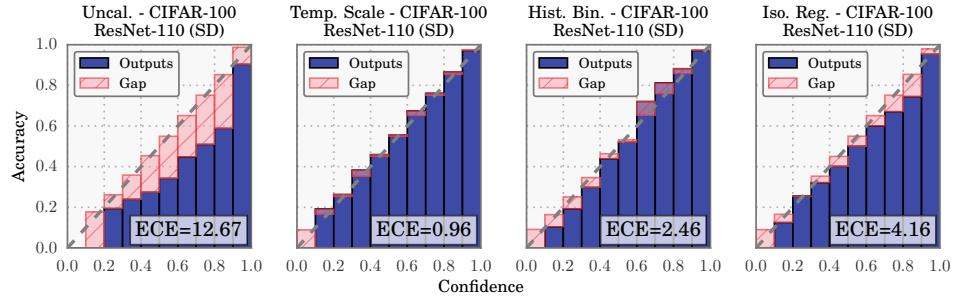


Figure 14.1: Reliability diagrams for the ResNet CNN image classifier [He+16b] applied to CIFAR-100 dataset. ECE is the expected calibration error, and measures the size of the red gap. Methods from left to right: original probabilities; after temperature scaling; after histogram binning; after isotonic regression. From Figure 4 of [Guo+17]. Used with kind permission of Chuan Guo.

More precisely, suppose we have B bins. Let \mathcal{B}_b be the set of indices of samples whose prediction confidence falls into the interval $I_b = (\frac{b-1}{B}, \frac{b}{B}]$. Here we use uniform bin widths, but we could also define the bins so that we can get an equal number of samples in each one.

Let $f(\mathbf{x})_c = p(y = c|\mathbf{x})$, $\hat{y}_n = \text{argmax}_{c \in \{1, \dots, C\}} f(\mathbf{x}_n)_c$, and $\hat{p}_n = \max_{c \in \{1, \dots, C\}} f(\mathbf{x}_n)_c$. The accuracy within bin b is defined as

$$\text{acc}(\mathcal{B}_b) = \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \mathbb{I}(\hat{y}_n = y_n) \quad (14.17)$$

The average confidence within this bin is defined as

$$\text{conf}(\mathcal{B}_b) = \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \hat{p}_n \quad (14.18)$$

If we plot accuracy vs confidence, we get a **reliability diagram**, as shown in Figure 14.1. The gap between the accuracy and confidence is shown in the red bars. We can measure this using the **expected calibration error (ECE)** [NCH15]:

$$\text{ECE}(f) = \sum_{b=1}^B \frac{|\mathcal{B}_b|}{B} |\text{acc}(\mathcal{B}_b) - \text{conf}(\mathcal{B}_b)| \quad (14.19)$$

In the multiclass case, the ECE only looks at the error of the MAP (top label) prediction. We can extend the metric to look at all the classes using the **marginal calibration error**, proposed in [KLM19]:

$$\text{MCE} = \sum_{c=1}^C w_c \mathbb{E} [(p(Y = c|f(\mathbf{x})_c) - f(\mathbf{x})_c)^2] \quad (14.20)$$

$$= \sum_{c=1}^C w_c \sum_{b=1}^B \frac{|\mathcal{B}_{b,c}|}{B} (\text{acc}(\mathcal{B}_{b,c}) - \text{conf}(\mathcal{B}_{b,c}))^2 \quad (14.21)$$

where $\mathcal{B}_{b,c}$ is the b 'th bin for class c , and $w_c \in [0, 1]$ denotes the importance of class c . (We can set $w_c = 1/C$ if all classes are equally important.) In [Nix+19], they call this metric **static calibration error**; they show that certain methods that have good ECE may have poor MCE. Other multi-class calibration metrics are discussed in [WLZ19].

14.2.2.2 Improving calibration

In principle, training a classifier so it optimizes a proper scoring rule (such as NLL) should automatically result in a well-calibrated classifier. In practice, however, unbalanced datasets can result in poorly calibrated predictions. Below we discuss various ways for improving the calibration of probabilistic classifiers, following [Guo+17].

14.2.2.3 Platt scaling

Let z be the log-odds, or logit, and $p = \sigma(z)$, produced by a probabilistic binary classifier. We wish to convert this to a more calibrated value q . The simplest way to do this is known as **Platt scaling**, and was proposed in [Pla00]. The idea is to compute $q = \sigma(az + b)$, where a and b are estimated via maximum likelihood on a validation set.

In the multiclass case, we can extend Platt scaling by using matrix scaling: $\mathbf{q} = \text{softmax}(\mathbf{W}\mathbf{z} + \mathbf{b})$, where we estimate \mathbf{W} and \mathbf{b} via maximum likelihood on a validation set. Since \mathbf{W} has $K \times K$ parameters, where K is the number of classes, this method can easily overfit, so in practice we restrict \mathbf{W} to be diagonal.

14.2.2.4 Nonparametric (histogram) methods

Platt scaling makes a strong assumption about how the shape of the calibration curve. A more flexible, nonparametric, method is to partition the predicted probabilities into bins, p_m , and to estimate an empirical probability q_m for each such bin; we then replace p_m with q_m ; this is known as **histogram binning** [ZE01a]. We can regularize this method by requiring that $q = f(p)$ be a piecewise constant, monotonically non-decreasing function; this is known as **isotonic regression** [ZE01a]. An alternative approach, known as the **scaling-binning calibrator**, is to apply a scaling method (such as Platt scaling), and then to apply histogram binning to that. This has the advantage of using the average of the scaled probabilities in each bin instead of the average of the observed binary labels (see Figure 14.2). In [KLM19], they prove that this results in better calibration, due to the lower variance of the estimator.

In the multiclass case, \mathbf{z} is the vector of logits, and $\mathbf{p} = \text{softmax}(\mathbf{z})$ is the vector of probabilities. We wish to convert this to a better calibrated version, \mathbf{q} . [ZE01b] propose to extend histogram binning and isotonic regression to this case by applying the above binary method to each of the K one-vs-rest problems, where K is the number of classes. However, this requires K separate calibration models, and results in an unnormalized probability distribution.

14.2.2.5 Temperature scaling

In [Guo+17], they noticed empirically that the diagonal version of Platt scaling, when applied to a variety of DNNs, often ended learning a vector of the form $\mathbf{w} = (c, c, \dots, c)$, for some constant c . This suggests a simpler form of scaling, which they call **temperature scaling**: $\mathbf{q} = \text{softmax}(\mathbf{z}/T)$,

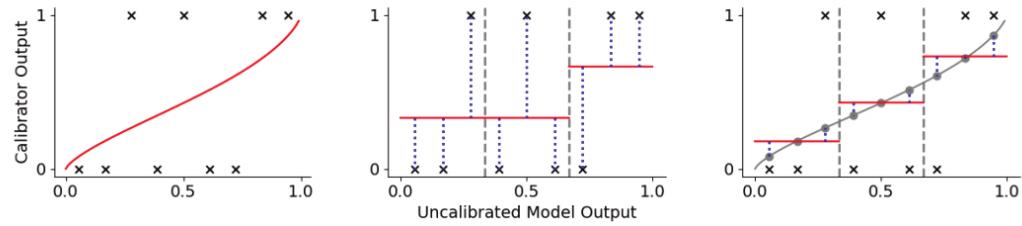


Figure 14.2: Visualization of 3 different approaches to calibrating a binary probabilistic classifier. Black crosses are the observed binary labels, red lines are the calibrated outputs. (a) Platt scaling. (b) Histogram binning with 3 bins. The output in each bin is the average of the binary labels in each bin. (c) The scaling-binning calibrator. This first applies Platt scaling, and then computes the average of the scaled points (gray circles) in each bin. From Figure 1 of [KLM19]. Used with kind permission of Ananya Kumar.

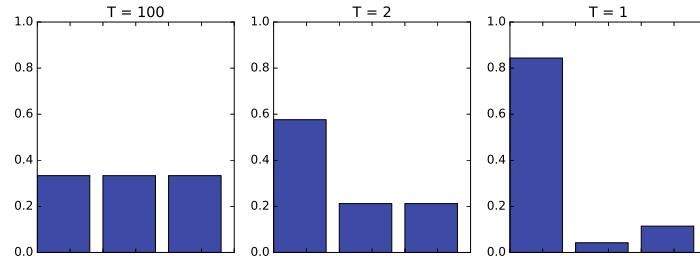


Figure 14.3: Softmax distribution $\text{softmax}(\mathbf{a}/T)$, where $\mathbf{a} = (3, 0, 1)$, at temperatures of $T = 100$, $T = 2$ and $T = 1$. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is “spiky”, with most of its mass on the largest element. Generated by softmax_plot.ipynb.

where $T > 0$ is a temperature parameter, which can be estimated by maximum likelihood on the validation set. The effect of this temperature parameter is to make the distribution less peaky, as shown in Figure 14.3. [Guo+17] show empirically that this method produces the lowest ECE on a variety of DNN classification problems (see Figure 14.1 for a visualization). Furthermore, it is much simpler and faster than the other methods.

Note that Platt scaling and temperature scaling do not affect the identity of the most probable class label, so these methods have no impact on classification accuracy. However, they do improve calibration performance. A more recent multi-class calibration method is discussed in [Kul+19].

14.2.2.6 Label smoothing

When training classifiers, we usually represent the true target label as a one-hot vector, say $\mathbf{y} = (0, 1, 0)$ to represent class 2 out of 3. We can improve results if we “spread” some of the probability mass across all the bins. For example we may use $\mathbf{y} = (0.1, 0.8, 0.1)$. This is called **label smoothing** and

often results in better-calibrated models [MKH19].

14.2.2.7 Bayesian methods

Bayesian approaches to fitting classifiers often result in more calibrated predictions, since they represent uncertainty in the parameters. See Section 17.3.8 for an example. However, [Ova+19] shows that well-calibrated models (even Bayesian ones) often become mis-calibrated when applied to inputs that come from a different distribution (see Section 19.2 for details).

14.2.3 Beyond evaluating marginal probabilities

Calibration (Section 14.2.2) focuses on assessing properties of the marginal predictive distribution $p(y|\mathbf{x})$. But this can sometimes be insufficient to distinguish between a good and bad model, especially in the context of online learning and sequential decision making, as pointed out in [Lu+22; Osb+21; WSG21; KKG22]. For example, consider two learning agents who observe a sequence of coin tosses. Let the outcome at time t be $Y_t \sim \text{Ber}(\theta)$, where θ is the unknown parameter. Agent 1 believes $\theta = 2/3$, whereas agent 2 believes either $\theta = 0$ or $\theta = 1$, but is not sure which, and puts probabilities $1/3$ and $2/3$ on these events. Thus both agents, despite having different models, make identical predictions for the next outcome: $p(Y_1^i = 0) = 1/3$ for agents $i = 1, 2$. However, the predictions of the two agents about a *sequence* of τ future outcomes is very different: In particular, agent 1 predicts each individual coin toss is a random Bernoulli event, where the probability is due to irreducible noise or **aleatoric uncertainty**:

$$p(Y_1^1 = 0, \dots, Y_\tau^1 = 0) = \frac{1}{3^\tau} \quad (14.22)$$

By contrast, agent 2 predicts that the sequence will either be all heads or all tails, where the probability is induced by **epistemic uncertainty** about the true parameters:

$$p(Y_1^2 = y_1, \dots, Y_\tau^2 = y_\tau) = \begin{cases} 1/3 & \text{if } y_1 = \dots = y_\tau = 0 \\ 2/3 & \text{if } y_1 = \dots = y_\tau = 1 \\ 0 & \text{otherwise} \end{cases} \quad (14.23)$$

The difference in beliefs between these agents will impact their behavior. For example, in a casino, agent 1 incurs little risk on repeatedly betting on heads in the long run, but for agent 2, this would be a very unwise strategy, and some initial information gathering (exploration) would be worthwhile.

Based on the above, we see that it is useful to evaluate *joint* predictive distributions when assessing predictive models. In [Lu+22; Osb+21] they propose to evaluate the posterior predictive distributions over τ outcomes $\mathbf{y} = Y_{T+1:T+\tau}$, given a set of τ inputs $\mathbf{x} = X_{T:T+\tau-1}$, and the past T data samples, $\mathcal{D}_T = \{(X_t, Y_{t+1}) : t = 0, 1, \dots, T-1\}$. The Bayes optimal predictive distribution is

$$P_T^B = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.24)$$

This is usually intractable to compute. Instead the agent will use an approximate distribution, known as a **belief state**, which we denote by

$$Q_T = p(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \quad (14.25)$$

The natural performance metric is the KL between these distributions. Since this depend on the inputs \mathbf{x} and $\mathcal{D}_T = (X_{0:T-1}, Y_{1:T})$, we will averaged the KL over these values, which are drawn iid from the true data generating distribution, which we denote by

$$P(X, Y, \mathcal{E}) = P(X|\mathcal{E})P(Y|X, \mathcal{E})P(\mathcal{E}) \quad (14.26)$$

where \mathcal{E} is the true but unknown environment. Thus we define our metric as

$$d_{B,Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T)} [D_{\text{KL}}(P^B(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.27)$$

where

$$P(\mathbf{x}, \mathcal{D}_T, \mathcal{E}) = P(\mathcal{E}) \underbrace{\left[\prod_{t=0}^{T-1} P(X_t|\mathcal{E})P(Y_{t+1}|X_t, \mathcal{E}) \right]}_{P(\mathcal{D}_T|\mathcal{E})} \underbrace{\left[\prod_{t=T}^{T+\tau-1} P(x_t|\mathcal{E}) \right]}_{P(\mathbf{x}|\mathcal{E})} \quad (14.28)$$

and $P(\mathbf{x}, \mathcal{D}_T)$ marginalizes this over environments.

Unfortunately, it is usually intractable to compute the exact Bayes posterior, P_T^B , so we cannot evaluate $d_{B,Q}^{KL}$. However, in Section 14.2.3.1, we show that

$$d_{B,Q}^{KL} = d_{\mathcal{E},Q}^{KL} - \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.29)$$

where the second term is a constant wrt the agent, and the first term is given by

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{E}) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.30)$$

$$= \mathbb{E}_{P(\mathbf{y}|\mathbf{x}, \mathcal{E})P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.31)$$

Hence if we rank agents in terms of $d_{\mathcal{E},Q}^{KL}$, it will give the same results as ranking them by $d_{B,Q}^{KL}$.

To compute $d_{\mathcal{E},Q}^{KL}$ in practice, we can use a Monte Carlo approximation: we just have to sample J environments, $\mathcal{E}^j \sim P(\mathcal{E})$, sample a training set \mathcal{D}_T from each environment, $\mathcal{D}_T^j \sim P(\mathcal{D}_T|\mathcal{E}^j)$, and then sample N data vectors of length τ , $(\mathbf{x}_n^j, \mathbf{y}_n^j) \sim P(X_{T:T+\tau-1}, Y_{T+1:T+\tau}|\mathcal{E}^j)$. We can then compute

$$\hat{d}_{\mathcal{E},Q}^{KL} = \frac{1}{JN} \sum_{j=1}^J \sum_{n=1}^N \left[\log P(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{E}^j) - \log Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{D}_T^j) \right] \quad (14.32)$$

where

$$p_{jn} = P(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{E}^j) = \prod_{t=T}^{T+\tau-1} P(Y_{n,t+1}^j|X_{n,t}^j, \mathcal{E}^j) \quad (14.33)$$

$$q_{jn} = Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \mathcal{D}_T^j) = \int Q(\mathbf{y}_n^j|\mathbf{x}_n^j, \boldsymbol{\theta})Q(\boldsymbol{\theta}|\mathcal{D}_T^j)d\boldsymbol{\theta} \quad (14.34)$$

$$\approx \frac{1}{M} \sum_{m=1}^M \prod_{t=T}^{T+\tau-1} Q(Y_{n,t+1}^j|X_{n,t}^j, \boldsymbol{\theta}_m^j) \quad (14.35)$$

where $\theta_m^j \sim Q(\theta|\mathcal{D}_T^j)$ is a sample from the agent's posterior over the environment.

The above assumes that $P(Y|X)$ is known; this will be the case if we use a synthetic data generator, as in the “neural testbed” in [Osb+21]. If we just have J empirical distributions for $P^j(X, Y)$, we can replace the KL with the cross entropy, which only differs by an additive constant:

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{E}) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] \quad (14.36)$$

$$= \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{E})} [\log P(\mathbf{y}|\mathbf{x}, \mathcal{E})]}_{\text{const}} - \underbrace{\mathbb{E}_{P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T|\mathcal{E})P(\mathcal{E})} [\log Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)]}_{d_{\mathcal{E},Q}^{CE}} \quad (14.37)$$

where the latter term is just the empirical negative log likelihood (NLL) of the agent on samples from the environment. Hence if we rank agents in terms of their NLL or cross entropy $d_{\mathcal{E},Q}^{CE}$ we will get the same results as ranking them by $d_{\mathcal{E},Q}^{KL}$, which will in turn give the same results as ranking them by $d_{B,Q}^{KL}$.

In practice we can approximate the cross entropy as follows:

$$\hat{d}_{\mathcal{E},Q}^{CE} = -\frac{1}{JN} \sum_{j=1}^J \sum_{n=1}^N \log Q(\mathbf{y}_n^j | \mathbf{x}_n^j, \mathcal{D}_T^j) \quad (14.38)$$

where $\mathcal{D}_T^j \sim P^j$, and $(\mathbf{x}_n^j, \mathbf{y}_n^j) \sim P^j$.

An alternative to estimating the KL or NLL is to evaluate the joint predictive accuracy by using it in a downstream task. In [Osb+21], they show that good predictive accuracy (for $\tau > 1$) correlates with good performance on a bandit problem (see Section 34.4). In [WSG21] they show that good predictive accuracy (for $\tau > 1$) results in good performance on a transductive active learning task.

14.2.3.1 Proof of claim

We now prove Equation (14.29), based on [Lu+21a]. First note that

$$d_{\mathcal{E},Q}^{KL} = \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T, \mathcal{E})P(\mathbf{y}|\mathbf{x}, \mathcal{E})} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.39)$$

$$= \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] + \mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] \quad (14.40)$$

For the first term in Equation (14.40) we have

$$\mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] = \sum P(\mathbf{x}, \mathbf{y}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.41)$$

$$= \sum P(\mathbf{x}, \mathcal{D}_T) \sum P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)}{Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \quad (14.42)$$

$$= \mathbb{E}_{P(\mathbf{x}, \mathcal{D}_T)} [D_{\text{KL}}(P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T) \| Q(\mathbf{y}|\mathbf{x}, \mathcal{D}_T))] = d_{B,Q}^{KL} \quad (14.43)$$

We now show that the second term in Equation (14.40) reduces to the mutual information. We exploit the fact that

$$P(\mathbf{y}|\mathbf{x}, \mathcal{E}) = P(\mathbf{y}|\mathcal{D}_T, \mathbf{x}, \mathcal{E}) = \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})}{P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})} \quad (14.44)$$



Figure 14.4: Prediction set examples on Imagenet. We show three progressively more difficult examples of the class fox squirrel and the prediction sets generated by conformal prediction. (Compare to Figure 17.9.) From Figure 1 of [AB21]. Used with kind permission of Anastasios Angelopoulos.

since \mathcal{D}_T has no new information in beyond \mathcal{E} . From this we get

$$\mathbb{E} \left[\log \frac{P(\mathbf{y}|\mathbf{x}, \mathcal{E})}{P(\mathbf{y}|\mathbf{x}, \mathcal{D}_T)} \right] = \mathbb{E} \left[\log \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})/P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})}{P(\mathbf{y}|\mathcal{D}, \mathcal{D}_T)} \right] \quad (14.45)$$

$$= \sum P(\mathcal{D}_T, \mathbf{x}) \sum P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \log \frac{P(\mathcal{E}, \mathbf{y}|\mathcal{D}_T, \mathbf{x})}{P(\mathbf{y}|\mathcal{D}_T, \mathbf{x})P(\mathcal{E}|\mathcal{D}_T, \mathbf{x})} \quad (14.46)$$

$$= \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.47)$$

Hence

$$d_{\mathcal{E}, Q}^{KL} = d_{B, Q}^{KL} + \mathbb{I}(\mathcal{E}; \mathbf{y}|\mathcal{D}_T, \mathbf{x}) \quad (14.48)$$

as claimed.

14.3 Conformal prediction

In this section, we briefly discuss **conformal prediction** [VGS05; SV08; ZFV20; AB21; KSB21; Man22b]. This is a simple but effective way to create prediction intervals or sets with guaranteed frequentist coverage probability from any predictive method $p(y|\mathbf{x})$. This can be seen as a form of **distribution free uncertainty quantification**, since it works without making assumptions (beyond exchangeability of the data) about the true data generating process or the form of the model.¹ Our presentation is based on the excellent tutorial of [AB21].²

In conformal prediction, we start with some heuristic notion of uncertainty — such as the softmax score for a classification problem, or the variance for a regression problem — and we use it to define a **conformal score** $s(\mathbf{x}, y) \in \mathbb{R}$, which measures how badly the output y “conforms” to \mathbf{x} . (Large

1. The exchangeability assumption rules out time series data, which is serially correlated. However, extensions to conformal prediction have been developed for the time series case, see e.g., [Zaf+22; Bha+23]. The exchangeability assumption also rules out distribution shift, although extensions to this case have also been developed [Tib+19].

2. See also the easy-to-use **MAPIE** Python library at <https://mapie.readthedocs.io/en/latest/index.html>, and the list of papers at [Man22a].

values of the score are less likely, so it is better to think of it as a non-conformity score.) Next we apply this score to a **calibration set** of n labeled examples, that was not used to train f , to get $\mathcal{S} = \{s_i = s(\mathbf{x}_i, y_i) : i = 1 : n\}$.³ The user specifies a desired confidence threshold α , say 0.1, and we then compute the $(1 - \alpha)$ quantile \hat{q} of \mathcal{S} . (In fact, we should replace $1 - \alpha$ with $\frac{\lceil(n+1)(1-\alpha)\rceil}{n}$, to account for the finite size of \mathcal{S} .) Finally, given a new test input, \mathbf{x}_{n+1} , we compute the prediction set to be

$$\mathcal{T}(\mathbf{x}_{n+1}) = \{y : s(\mathbf{x}_{n+1}, y) \leq \hat{q}\} \quad (14.49)$$

Intuitively, we include all the outputs y that are plausible given the input. See Figure 14.4 for an illustration.

Remarkably, one can show the following general result

$$1 - \alpha \leq P^*(y^{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})) \leq 1 - \alpha + \frac{1}{n+1} \quad (14.50)$$

where the probability is wrt the true distribution $P^*(\mathbf{x}_1, y_1, \dots, \mathbf{x}_{n+1}, y_{n+1})$. We say that the prediction set has a **coverage** level of $1 - \alpha$. This holds for any value of $n \geq 1$ and $\alpha \in [0, 1]$. The only assumption is that the values (\mathbf{x}_i, y_i) are exchangeable, and hence the calibration scores s_i are also exchangeable.

To see why this is true, let us sort the scores so $s_1 < \dots < s_n$, so $\hat{q} = s_i$, where $i = \frac{\lceil(n+1)(1-\alpha)\rceil}{n}$. (We assume the scores are distinct, for simplicity.) The score s_{n+1} is equally likely to fall in anywhere between the calibration points s_1, \dots, s_n , since the points are exchangeable. Hence

$$P^*(s_{n+1} \leq s_k) = \frac{k}{n+1} \quad (14.51)$$

for any $k \in \{1, \dots, n+1\}$. The event $\{y_{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})\}$ is equivalent to $\{s_{n+1} \leq \hat{q}\}$. Hence

$$P^*(y_{n+1} \in \mathcal{T}(\mathbf{x}_{n+1})) = P^*(s_{n+1} \leq \hat{q}) = \frac{\lceil(n+1)(1-\alpha)\rceil}{n+1} \geq 1 - \alpha \quad (14.52)$$

For the proof of the upper bound, see [Lei+18].

Although this result may seem like a “free lunch”, it is worth noting that we can always achieve a desired coverage level by defining the prediction set to be all possible labels. In this case, the prediction set will be independent of the input, but it will cover the true label $1 - \alpha$ of the time. To rule out such degenerate cases, we seek prediction sets that are as small as possible (although we allow for the set to be larger for harder examples), while meeting the coverage requirement. Achieving this goal requires that we define suitable conformal scores. Below we give some examples of how to compute conformal scores $s(\mathbf{x}, y)$ for different kinds of problem.⁴ It is also important to note that the coverage guarantees are frequentist in nature, and refer to average behavior, rather than representing per-instance uncertainty, as in the Bayesian approach.

3. Using a calibration set is called **split conformal prediction**. If we don’t have enough data to adopt this splitting approach, we can use **full conformal prediction** [VGS05], which requires fitting the model n times using a leave-one-out type procedure.

4. It is also possible to learn conformal scores in an end-to-end way, jointly with the predictive model, as discussed in [Stu+22].

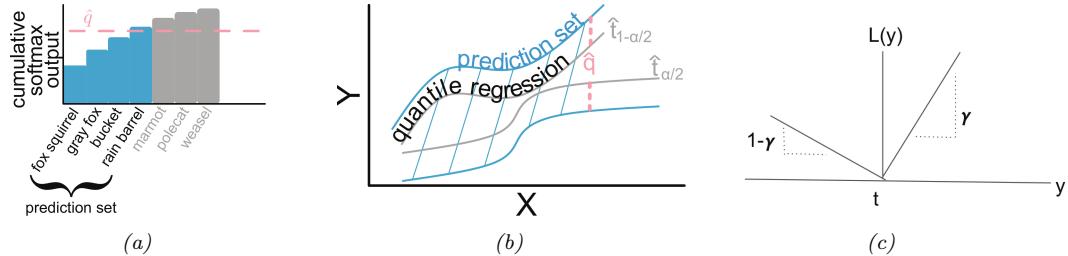


Figure 14.5: (a) Illustration of adaptive prediction set. From Figure 5 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (b) Illustrate of conformalized quantile regression. From Figure 6 of [AB21]. Used with kind permission of Anastasios Angelopoulos. (c) Illustration of pinball loss function.

14.3.1 Conformalizing classification

The simplest way to apply conformal prediction to multiclass classification is to derive the conformal score from the softmax score assigned to the label using $s(\mathbf{x}, y) = 1 - f(\mathbf{x})_y$, so large values are considered less likely than small values. We compute the threshold \hat{q} as described above, and then we define the prediction set to be $\mathcal{T}(\mathbf{x}) = \{y : f(\mathbf{x})_y \geq 1 - \hat{q}\}$, which matches Equation (14.49). That is, we take the set of all class labels above the specified threshold, as illustrated in Figure 14.4.

Although the above approach produces prediction sets with the smallest average size (as proved in [SLW19]), the size of the set tends to be too large for easy examples and too small for hard examples. We now present an improved method, known as **adaptive prediction sets**, due to [RSC20], which solves this problem. The idea is simple: we sort all the softmax scores, $f(\mathbf{x})_c$ for $c = 1 : C$, to get permutation $\pi_{1:C}$, and then we define $s(\mathbf{x}, y)$ to be the cumulative sum of the scores up until we reach label y : $s(\mathbf{x}, y) = \sum_{c=1}^k f(\mathbf{x})_{\pi_c}$, where $k = \pi_y$. We now compute \hat{q} as before, and define the prediction set $\mathcal{T}(\mathbf{x})$ to be the set of all labels, sorted in order of decreasing probability, until we cover \hat{q} of the probability mass. See Figure 14.5a for an illustration. This uses all the softmax scores output by the model, rather than just the top score, which accounts for its improved performance.

14.3.2 Conformalizing regression

In this section, we consider conformalized regression problems. Since now $y \in \mathbb{R}$, computing the prediction set in Equation (14.49) is expensive, so instead we will compute a prediction interval, specified by a lower and upper bound.

14.3.2.1 Conformalizing quantile regression

In this section, we use **quantile regression** to compute the lower and upper bounds. We first fit a function of the form $t_\gamma(\mathbf{x})$, which predicts the γ quantile of the cdf $P_Y(Y) = p(Y|\mathbf{x})$. For example, if we set $\gamma = 0.5$, we get the median. If we use $\gamma = 0.05$ and $\gamma = 0.95$, we can get an approximate 90% prediction interval using $[t_{0.05}(\mathbf{x}), t_{0.95}(\mathbf{x})]$, as illustrated by the gray lines in Figure 14.5b.

To fit the quantile regression model, we just replace squared loss with the **quantile loss**, also called the **pinball loss**, which is defined as

$$\ell_\gamma(y, \hat{t}) = (\gamma - \mathbb{I}(y < \hat{t}))(y - \hat{t}) = (y - \hat{t})\gamma\mathbb{I}(y > \hat{t}) + (\hat{t} - y)(1 - \gamma)\mathbb{I}(y < \hat{t}) \quad (14.53)$$

where y is the true output and \hat{t} is the predicted value at quantile γ . See Figure 14.5c for an illustration. The key property of this loss function is that its minimizer is the γ -quantile of the distribution P_Y , i.e.,

$$\underset{\hat{t}}{\operatorname{argmin}} \mathbb{E}_{P_Y(y)} [\ell_\gamma(y, \hat{t})] = P_Y^{-1}(\gamma)$$

However, the regression quantiles are usually only approximately a 90% interval because the model may be mismatched to the true distribution. Fortunately we can use conformal prediction to fix this. In particular, let us define the conformal score to be

$$s(\mathbf{x}, y) = \max(\hat{t}_{\alpha/2}(\mathbf{x}) - y, y - \hat{t}_{\alpha/2}(\mathbf{x})) \quad (14.54)$$

In other words, $s(\mathbf{x}, y)$ is a positive measure of how far the value y is outside the prediction interval, or is a negative measure if y is inside the prediction interval. We compute \hat{q} as before, and define the conformal prediction interval to be

$$\mathcal{T}(\mathbf{x}) = [\hat{t}_{\alpha/2}(\mathbf{x}) - \hat{q}, \hat{t}_{\alpha/2}(\mathbf{x}) + \hat{q}] \quad (14.55)$$

This makes the quantile regression interval wider if \hat{q} is positive (if the base method was overconfident), and narrower if \hat{q} is negative (if the base method was underconfident). See Figure 14.5b for an illustration. This approach is called **conformalized quantile regression** or **CQR** [RPC19].

14.3.2.2 Conformalizing predicted variances

There are many ways to define uncertainty scores $u(\mathbf{x})$, such as the predicted standard deviation, from which we can derive a prediction interval using

$$\mathcal{T}(\mathbf{x}) = [f(\mathbf{x}) - u(\mathbf{x})\hat{q}, f(\mathbf{x}) + u(\mathbf{x})\hat{q}] \quad (14.56)$$

Here \hat{q} is derived from the quantiles of the following conformal scores

$$s(\mathbf{x}, y) = \frac{|y - f(\mathbf{x})|}{u(\mathbf{x})} \quad (14.57)$$

The interval produced by this method tends to be wider than the one computed by CQR, since it extends an equal amount above and below the predicted value $f(\mathbf{x})$. In addition, the uncertainty measure $u(\mathbf{x})$ may not scale properly with α . Nevertheless, this is a simple post-hoc method that can be applied to many regression methods without needing to retrain them.

15 Generalized linear models

15.1 Introduction

A **generalized linear model** or **GLM** [MN89] is a conditional version of an exponential family distribution (Section 2.4). More precisely, the model has the following form:

$$p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \exp \left[\frac{y_n \eta_n - A(\eta_n)}{\sigma^2} + \log h(y_n, \sigma^2) \right] \quad (15.1)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$ is the natural parameter for the distribution, $A(\eta_n)$ is the log normalizer, $\mathcal{T}(y) = y$ is the sufficient statistic, and σ^2 is the dispersion term. Based on the results in Section 2.4.3, we can show that the mean and variance of the response variable are as follows:

$$\mu_n \triangleq \mathbb{E}[y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2] = A'(\eta_n) \triangleq \ell^{-1}(\eta_n) \quad (15.2)$$

$$\mathbb{V}[y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2] = A''(\eta_n) \sigma^2 \quad (15.3)$$

We will denote the mapping from the linear inputs to the mean of the output using $\mu_n = \ell^{-1}(\eta_n)$, where the function ℓ is known as the **link function**, and ℓ^{-1} is known as the **mean function**. This relationship is usually written as follows:

$$\ell(\mu_n) = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.4)$$

GLMs are quite limited in their predictive power, due to the assumption of linearity (although we can always use basis function expansion on \mathbf{x}_n to improve the flexibility). However, the main use of GLMs in the statistics literature is not for prediction, but for hypothesis testing, as we explain in Section 3.10.3. This relies on the ability to compute the posterior, $p(\mathbf{w} | \mathcal{D})$, which we discuss in Section 15.1.4. We can use this to draw conclusions about whether any of the inputs (e.g., representing different groups) have a significant effect on the output.

15.1.1 Some popular GLMs

In this section, we give some examples of widely used GLMs.

15.1.1.1 Linear regression

Recall that linear regression has the form

$$p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right) \quad (15.5)$$

Hence

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2}(y_n - \eta_n)^2 - \frac{1}{2}\log(2\pi\sigma^2) \quad (15.6)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. We can write this in GLM form as follows:

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}, \sigma^2) = \frac{y_n \eta_n - \frac{\eta_n^2}{2}}{\sigma^2} - \frac{1}{2} \left(\frac{y_n^2}{\sigma^2} + \log(2\pi\sigma^2) \right) \quad (15.7)$$

We see that $A(\eta_n) = \eta_n^2/2$ and hence

$$\mathbb{E}[y_n] = \eta_n = \mathbf{w}^\top \mathbf{x}_n \quad (15.8)$$

$$\mathbb{V}[y_n] = \sigma^2 \quad (15.9)$$

See Section 15.2 for details on linear regression.

15.1.1.2 Binomial regression

If the response variable is the number of successes in N_n trials, $y_n \in \{0, \dots, N_n\}$, we can use **binomial regression**, which is defined by

$$p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = \text{Bin}(y_n | \sigma(\mathbf{w}^\top \mathbf{x}_n), N_n) \quad (15.10)$$

We see that binary logistic regression is the special case when $N_n = 1$.

The log pdf is given by

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \log \mu_n + (N_n - y_n) \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.11)$$

$$= y_n \log\left(\frac{\mu_n}{1 - \mu_n}\right) + N_n \log(1 - \mu_n) + \log \binom{N_n}{y_n} \quad (15.12)$$

where $\mu_n = \sigma(\eta_n)$. To rewrite this in GLM form, let us define

$$\eta_n \triangleq \log \left[\frac{\mu_n}{(1 - \mu_n)} \right] = \log \left[\frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}} \frac{1 + e^{-\mathbf{w}^\top \mathbf{x}_n}}{e^{-\mathbf{w}^\top \mathbf{x}_n}} \right] = \log \frac{1}{e^{-\mathbf{w}^\top \mathbf{x}_n}} = \mathbf{w}^\top \mathbf{x}_n \quad (15.13)$$

Hence we can write binomial regression in GLM form as follows

$$\log p(y_n | \mathbf{x}_n, N_n, \mathbf{w}) = y_n \eta_n - A(\eta_n) + h(y_n) \quad (15.14)$$

where $h(y_n) = \log \binom{N_n}{y_n}$ and

$$A(\eta_n) = -N_n \log(1 - \mu_n) = N_n \log(1 + e^{\eta_n}) \quad (15.15)$$

Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = \frac{N_n e^{\eta_n}}{1 + e^{\eta_n}} = \frac{N_n}{1 + e^{-\eta_n}} = N_n \mu_n \quad (15.16)$$

and

$$\mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = N_n \mu_n (1 - \mu_n) \quad (15.17)$$

See Section 15.3.9 for an example of binomial regression.

15.1.1.3 Poisson regression

If the response variable is an integer count, $y_n \in \{0, 1, \dots\}$, we can use **Poisson regression**, which is defined by

$$p(y_n | \mathbf{x}_n, \mathbf{w}) = \text{Poi}(y_n | \exp(\mathbf{w}^\top \mathbf{x}_n)) \quad (15.18)$$

where

$$\text{Poi}(y | \mu) = e^{-\mu} \frac{\mu^y}{y!} \quad (15.19)$$

is the Poisson distribution. Poisson regression is widely used in bio-statistical applications, where y_n might represent the number of diseases of a given person or place, or the number of reads at a genomic location in a high-throughput sequencing context (see e.g., [Kua+09]).

The log pdf is given by

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}) = y_n \log \mu_n - \mu_n - \log(y_n!) \quad (15.20)$$

where $\mu_n = \exp(\mathbf{w}^\top \mathbf{x}_n)$. Hence in GLM form we have

$$\log p(y_n | \mathbf{x}_n, \mathbf{w}) = y_n \eta_n - A(\eta_n) + h(y_n) \quad (15.21)$$

where $\eta_n = \log(\mu_n) = \mathbf{w}^\top \mathbf{x}_n$, $A(\eta_n) = \mu_n = e^{\eta_n}$, and $h(y_n) = -\log(y_n!)$. Hence

$$\mathbb{E}[y_n] = \frac{dA}{d\eta_n} = \frac{d}{d\eta_n} e^{\eta_n} = \mu_n \quad (15.22)$$

and

$$\mathbb{V}[y_n] = \frac{d^2 A}{d\eta_n^2} = \frac{d}{d\eta_n} \frac{d}{d\eta_n} e^{\eta_n} = e^{\eta_n} = \mu_n \quad (15.23)$$

15.1.1.4 Zero-inflated Poisson regression

In many forms of count data, the number of observed 0s is larger than what a model might expect, even after taking into account the predictors. Intuitively, this is because there may be many ways to produce no outcome. For example, consider predicting sales data for a product. If the sales are 0, does it mean the product is unpopular (so the demand is very low), or was it simply sold out (implying the demand is high, but supply is zero)? Similar problems arise in genomics, epidemiology, etc.

To handle such situations, it is common to use a **zero-inflated Poisson** or **ZIP** model. The likelihood for this model is a mixture of two distributions: a spike at 0, and a standard Poisson. Formally, we define

$$\text{ZIP}(y|\rho, \lambda) = \begin{cases} \rho + (1 - \rho) \exp(-\lambda) & \text{if } y = 0 \\ (1 - \rho) \frac{\lambda^y \exp(-\lambda)}{y!} & \text{if } y > 0 \end{cases} \quad (15.24)$$

Here ρ is the prior probability of picking the spike, and λ is the rate of the Poisson. We see that there are two “mechanisms” for generating a 0: either (with probability ρ) we choose the spike, or (with probability $1 - \rho$) we simply generate a zero count just because the rate of the Poisson is so low. (This latter event has probability $\lambda^0 e^{-\lambda} / 0! = e^{-\lambda}$.)

15.1.2 GLMs with noncanonical link functions

We have seen how the mean parameters of the output distribution are given by $\mu = \ell^{-1}(\eta)$, where the function ℓ is the link function. There are several choices for this function, as we now discuss.

The **canonical link function** ℓ satisfies the property that $\theta = \ell(\mu)$, where θ are the canonical (natural) parameters. Hence

$$\theta = \ell(\mu) = \ell(\ell^{-1}(\eta)) = \eta \quad (15.25)$$

This is what we have assumed so far. For example, for the Bernoulli distribution, the canonical parameter is the log-odds $\eta = \log(\mu/(1 - \mu))$, which is given by the logit transform

$$\eta = \ell(\mu) = \text{logit}(\mu) = \log\left(\frac{\mu}{1 - \mu}\right) \quad (15.26)$$

The inverse of this is the sigmoid or logistic function

$$\mu = \ell^{-1}(\eta) = \sigma(\eta) = 1/(1 + e^{-\eta}) \quad (15.27)$$

However, we are free to use other kinds of link function. For example, in Section 15.4 we use

$$\eta = \ell(\mu) = \Phi^{-1}(\mu) \quad (15.28)$$

$$\mu = \ell^{-1}(\eta) = \Phi(\eta) \quad (15.29)$$

This is known as the **probit link function**.

Another link function that is sometimes used for binary responses is the **complementary log-log** function

$$\eta = \ell(\mu) = \log(-\log(1 - \mu)) \quad (15.30)$$

This is used in applications where we either observe 0 events (denoted by $y = 0$) or one or more (denoted by $y = 1$), where events are assumed to be governed by a Poisson distribution with rate λ . Let E be the number of events. The Poisson assumption means $p(E = 0) = \exp(-\lambda)$ and hence

$$p(y = 0) = (1 - \mu) = p(E = 0) = \exp(-\lambda) \quad (15.31)$$

Thus $\lambda = -\log(1 - \mu)$. When λ is a function of covariates, we need to ensure it is positive, so we use $\lambda = e^\eta$, and hence

$$\eta = \log(\lambda) = \log(-\log(1 - \mu)) \quad (15.32)$$

15.1.3 Maximum likelihood estimation

GLMs can be fit using similar methods to those that we used to fit logistic regression. In particular, the negative log-likelihood has the following form (ignoring constant terms):

$$\text{NLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) = -\frac{1}{\sigma^2} \sum_{n=1}^N \ell_n \quad (15.33)$$

where

$$\ell_n \triangleq \eta_n y_n - A(\eta_n) \quad (15.34)$$

where $\eta_n = \mathbf{w}^\top \mathbf{x}_n$. For notational simplicity, we will assume $\sigma^2 = 1$.

We can compute the gradient for a single term as follows:

$$\mathbf{g}_n \triangleq \frac{\partial \ell_n}{\partial \mathbf{w}} = \frac{\partial \ell_n}{\partial \eta_n} \frac{\partial \eta_n}{\partial \mathbf{w}} = (y_n - A'(\eta_n)) \mathbf{x}_n = (y_n - \mu_n) \mathbf{x}_n \quad (15.35)$$

where $\mu_n = f(\mathbf{w}^\top \mathbf{x}_n)$, and f is the inverse link function that maps from canonical parameters to mean parameters. (For example, in the case of logistic regression, we have $\mu_n = \sigma(\mathbf{w}^\top \mathbf{x})$.)

The Hessian is given by

$$\mathbf{H} = \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}^\top} \text{NLL}(\mathbf{w}) = - \sum_{n=1}^N \frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} \quad (15.36)$$

where

$$\frac{\partial \mathbf{g}_n}{\partial \mathbf{w}^\top} = \frac{\partial \mathbf{g}_n}{\partial \mu_n} \frac{\partial \mu_n}{\partial \mathbf{w}^\top} = -\mathbf{x}_n f'(\mathbf{w}^\top \mathbf{x}_n) \mathbf{x}_n^\top \quad (15.37)$$

Hence

$$\mathbf{H} = \sum_{n=1}^N f'(\eta_n) \mathbf{x}_n \mathbf{x}_n^\top \quad (15.38)$$

For example, in the case of logistic regression, $f(\eta_n) = \sigma(\eta_n) = \mu_n$, and $f'(\eta_n) = \mu_n(1 - \mu_n)$. In general, we see that the Hessian is positive definite, since $f'(\eta_n) > 0$; hence the negative log likelihood is convex, so the MLE for a GLM is unique (assuming $f(\eta_n) > 0$ for all n).

For small datasets, we can use the **iteratively reweighted least squares** or **IRLS** algorithm, which is a form of Newton's method, to compute the MLE (see e.g., [Mur22, Sec 10.2.6]). For large datasets, we can use SGD. (In practice it is often useful to combine SGD with methods that automatically tune the step size, such as [Loi+21].)

15.1.4 Bayesian inference

Maximum likelihood estimation provides a point estimate of the parameters, but does not convey any notion of uncertainty, which is important for hypothesis testing, as we explain in Section 3.10.3,

as well as for avoiding overfitting. To compute the uncertainty, we will perform Bayesian inference of the parameters. To do this, we first need to specify a prior. Choosing a suitable prior depends on the form of link function. For example, a “flat” or “uninformative” prior on the offset term $\alpha \in \mathbb{R}$ will not translate to an uninformative prior on the probability scale if we pass α through a sigmoid, as we discuss in Section 15.3.4.

Once we have chosen the prior, we can compute the posterior using a variety of approximate inference methods. For small datasets, HMC (Section 12.5) is the easiest to use, since you just need to write down the log likelihood and log prior; we can then use autograd to compute derivatives which can be passed to the HMC engine (see e.g., [BG13] for details).

There are many standard software packages for HMC analysis of (hierarchical) GLMs, such as **Bambi** (<https://github.com/bambinos/bambi>), which is a Python wrapper on top of PyMC/BlackJAX, **RStanARM** (<https://cran.r-project.org/web/packages/rstanarm/index.html>), which is an R wrapper on top of Stan, and **BRMS** (<https://cran.r-project.org/web/packages/brms/index.html>), which is another R wrapper on top of Stan. These libraries support a convenient **formula syntax**, initially created in the R language, for compactly specifying the form of the model, including possible interaction terms between the inputs.

For large datasets, HMC can be slow, since it is a full batch algorithm. In such settings, variational Bayes (see e.g., [HOW11; TN13]), expectation propagation (see e.g., [KW18]), or more specialized algorithms (e.g., [HAB17]) are the best choice.

15.2 Linear regression

Linear regression is the simplest case of a GLM, and refers to the following model:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + \mathbf{w}^\top \mathbf{x}, \sigma^2) \quad (15.39)$$

where $\boldsymbol{\theta} = (w_0, \mathbf{w}, \sigma^2)$ are all the parameters of the model. (In statistics, the parameters w_0 and \mathbf{w} are usually denoted by β_0 and $\boldsymbol{\beta}$.) We gave a detailed introduction to this model in the prequel to this book, [Mur22]. In this section, we briefly discuss maximum likelihood estimation, and then focus on a Bayesian analysis.

15.2.1 Ordinary least squares

From Equation (15.39), we can derive the negative log likelihood of the data as follows:

$$\text{NLL}(\mathbf{w}, \sigma^2) = -\sum_{n=1}^N \log \left[\left(\frac{1}{2\pi\sigma^2} \right)^{\frac{1}{2}} \exp \left(-\frac{1}{2\sigma^2} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 \right) \right] \quad (15.40)$$

$$= \frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \hat{y}_n)^2 + \frac{N}{2} \log(2\pi\sigma^2) \quad (15.41)$$

where we have defined the predicted response $\hat{y}_n \triangleq \mathbf{w}^\top \mathbf{x}_n$. In [Mur22, Sec 11.2.2] we show that the MLE is given by

$$\hat{\mathbf{w}}_{\text{mle}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.42)$$

This is called the **ordinary least squares (OLS)** solution.

The MLE for the observation noise is given by

$$\hat{\sigma}_{\text{mle}}^2 = \underset{\sigma^2}{\operatorname{argmin}} \text{NLL}(\hat{\mathbf{w}}, \sigma^2) = \frac{1}{N} \sum_{n=1}^N (y_n - \mathbf{x}_n^\top \hat{\mathbf{w}})^2 \quad (15.43)$$

This is just the mean squared error of the residuals, which is an intuitive result.

15.2.2 Conjugate priors

In this section, we derive the posterior for the parameters using a conjugate prior. We first consider the case where just \mathbf{w} is unknown (so the observation noise variance parameter σ^2 is fixed), and then we consider the general case, where both σ^2 and \mathbf{w} are unknown.

15.2.2.1 Noise variance is known

The conjugate prior for linear regression has the following form:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \tilde{\Sigma}) \quad (15.44)$$

We often use $\tilde{\mathbf{w}} = \mathbf{0}$ as the prior mean and $\tilde{\Sigma} = \tau^2 \mathbf{I}_D$ as the prior covariance. (We assume the bias term is included in the weight vector, but often use a much weaker prior for it, since we typically do not want to regularize the overall mean level of the output.)

To derive the posterior, let us first rewrite the likelihood in terms of an MVN as follows:

$$\ell(\mathbf{w}) = p(\mathcal{D} | \mathbf{w}, \sigma^2) = \prod_{n=1}^N p(y_n | \mathbf{w}^\top \mathbf{x}_n, \sigma^2) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \quad (15.45)$$

where \mathbf{I}_N is the $N \times N$ identity matrix. We can then use Bayes' rule for Gaussians (Equation (2.121)) to derive the posterior, which is as follows:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}, \sigma^2) \propto \mathcal{N}(\mathbf{w} | \tilde{\mathbf{w}}, \tilde{\Sigma}) \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) = \mathcal{N}(\mathbf{w} | \hat{\mathbf{w}}, \hat{\Sigma}) \quad (15.46)$$

$$\hat{\mathbf{w}} \triangleq \hat{\Sigma}^{-1} (\tilde{\Sigma}^{-1} \tilde{\mathbf{w}} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{y}) \quad (15.47)$$

$$\hat{\Sigma} \triangleq (\tilde{\Sigma}^{-1} + \frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.48)$$

where $\hat{\mathbf{w}}$ is the posterior mean, and $\hat{\Sigma}$ is the posterior covariance.

Now suppose $\tilde{\mathbf{w}} = \mathbf{0}$ and $\tilde{\Sigma} = \tau^2 \mathbf{I}$. In this case, the posterior mean becomes

$$\hat{\mathbf{w}} = \frac{1}{\sigma^2} \hat{\Sigma} \mathbf{X}^\top \mathbf{y} = (\frac{\sigma^2}{\tau^2} \mathbf{I} + \mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.49)$$

If we define $\lambda = \frac{\sigma^2}{\tau^2}$, we see this is equivalent to **ridge regression**, which optimizes

$$\mathcal{L}(\mathbf{w}) = \text{RSS}(\mathbf{w}) + \lambda \|\mathbf{w}\|^2 \quad (15.50)$$

where RSS is the residual sum of squares:

$$\text{RSS}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 = \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \frac{1}{2} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \quad (15.51)$$

15.2.2.2 Noise variance is unknown

In this section, we assume \mathbf{w} and σ^2 are both unknown. The likelihood is given by

$$\ell(\mathbf{w}, \sigma^2) = p(\mathcal{D}|\mathbf{w}, \sigma^2) \propto (\sigma^2)^{-N/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2\right) \quad (15.52)$$

Since the regression weights now depend on σ^2 in the likelihood, the conjugate prior for \mathbf{w} has the form

$$p(\mathbf{w}|\sigma^2) = \mathcal{N}(\mathbf{w}|\tilde{\mathbf{w}}, \sigma^2 \tilde{\Sigma}) \quad (15.53)$$

For the noise variance σ^2 , the conjugate prior is based on the inverse gamma distribution, which has the form

$$\text{IG}(\sigma^2|\tilde{a}, \tilde{b}) = \frac{\tilde{b}^{\tilde{a}}}{\Gamma(\tilde{a})} (\sigma^2)^{-(\tilde{a}+1)} \exp(-\frac{\tilde{b}}{\sigma^2}) \quad (15.54)$$

(See Section 2.2.3.4 for more details.) Putting these two together, we find that the joint conjugate prior is the **normal inverse gamma** distribution:

$$\text{NIG}(\mathbf{w}, \sigma^2|\tilde{\mathbf{w}}, \tilde{\Sigma}, \tilde{a}, \tilde{b}) \triangleq \mathcal{N}(\mathbf{w}|\tilde{\mathbf{w}}, \sigma^2 \tilde{\Sigma}) \text{IG}(\sigma^2|\tilde{a}, \tilde{b}) \quad (15.55)$$

$$\begin{aligned} &= \frac{\tilde{b}^{\tilde{a}}}{(2\pi)^{D/2} |\tilde{\Sigma}|^{1/2} \Gamma(\tilde{a})} (\sigma^2)^{-(\tilde{a}+(D/2)+1)} \\ &\times \exp\left[-\frac{(\mathbf{w}-\tilde{\mathbf{w}})^\top \tilde{\Sigma}^{-1} (\mathbf{w}-\tilde{\mathbf{w}}) + 2\tilde{b}}{2\sigma^2}\right] \end{aligned} \quad (15.56)$$

This results in the following posterior:

$$p(\mathbf{w}, \sigma^2|\mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2|\hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.57)$$

$$\hat{\mathbf{w}} = \hat{\Sigma}^{-1} \tilde{\mathbf{w}} + \mathbf{X}^\top \mathbf{y} \quad (15.58)$$

$$\hat{\Sigma} = (\tilde{\Sigma}^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \quad (15.59)$$

$$\hat{a} = \tilde{a} + N/2 \quad (15.60)$$

$$\hat{b} = \tilde{b} + \frac{1}{2} \left(\tilde{\mathbf{w}}^\top \tilde{\Sigma}^{-1} \tilde{\mathbf{w}} + \mathbf{y}^\top \mathbf{y} - \hat{\mathbf{w}}^\top \hat{\Sigma}^{-1} \hat{\mathbf{w}} \right) \quad (15.61)$$

The expressions for $\hat{\mathbf{w}}$ and $\hat{\Sigma}$ are similar to the case where σ^2 is known. The expression for \hat{a} is also intuitive, since it just updates the counts. The expression for \hat{b} can be interpreted as follows: it is the prior sum of squares, \tilde{b} , plus the empirical sum of squares, $\mathbf{y}^\top \mathbf{y}$, plus a term due to the error in the prior on \mathbf{w} .

The posterior marginals are as follows. For the variance, we have

$$p(\sigma^2|\mathcal{D}) = \int p(\mathbf{w}|\sigma^2, \mathcal{D}) p(\sigma^2|\mathcal{D}) d\mathbf{w} = \text{IG}(\sigma^2|\hat{a}, \hat{b}) \quad (15.62)$$

For the regression weights, it can be shown that

$$p(\mathbf{w}|\mathcal{D}) = \int p(\mathbf{w}|\sigma^2, \mathcal{D}) p(\sigma^2|\mathcal{D}) d\sigma^2 = \mathcal{T}(\mathbf{w}|\hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}} \hat{\Sigma}, 2\hat{a}) \quad (15.63)$$

15.2.2.3 Posterior predictive distribution

In machine learning we usually care more about uncertainty (and accuracy) of our predictions, not our parameter estimates. Fortunately, one can derive the posterior predictive distribution in closed form. In particular, one can show that, given N' new test inputs $\tilde{\mathbf{X}}$, we have

$$p(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}, \mathcal{D}) = \int \int p(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}, \mathbf{w}, \sigma^2) p(\mathbf{w}, \sigma^2 | \mathcal{D}) d\mathbf{w} d\sigma^2 \quad (15.64)$$

$$= \int \int \mathcal{N}(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}\mathbf{w}, \sigma^2 \mathbf{I}_{N'}) \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) d\mathbf{w} d\sigma^2 \quad (15.65)$$

$$= \mathcal{T}(\tilde{\mathbf{y}}|\tilde{\mathbf{X}}\hat{\mathbf{w}}, \frac{\hat{b}}{\hat{a}}(\mathbf{I}_{N'} + \tilde{\mathbf{X}}\hat{\Sigma}\tilde{\mathbf{X}}^\top), 2\hat{a}) \quad (15.66)$$

The posterior predictive mean is equivalent to “normal” linear regression, but where we plug in $\hat{\mathbf{w}} = \mathbb{E}[\mathbf{w}|\mathcal{D}]$ instead of the MLE. The posterior predictive variance has two components: $\hat{b}/\hat{a}\mathbf{I}_{N'}$ due to the measurement noise, and $\hat{b}/\hat{a}\tilde{\mathbf{X}}\hat{\Sigma}\tilde{\mathbf{X}}^\top$ due to the uncertainty in \mathbf{w} . This latter term varies depending on how close the test inputs are to the training data. The results are similar to using a Gaussian prior (with fixed σ^2), except the predictive distribution is even wider, since we are taking into account uncertainty about σ^2 .

15.2.3 Uninformative priors

A common criticism of Bayesian inference is the need to use a prior. This is sometimes thought to “pollute” the inferences one makes from the data. We can minimize the effect of the prior by using an uninformative prior, as we discussed in Section 3.5. Below we discuss various uninformative priors for linear regression.

15.2.3.1 Jeffreys prior

From Section 3.5.3.1, we know that the Jeffreys prior for the location parameter has the form $p(\mathbf{w}) \propto 1$, and from Section 3.5.3.2, we know that the Jeffreys prior for the scale factor has the form $p(\sigma) \propto \sigma^{-1}$. We can emulate these priors using an improper NIG prior with $\check{\mathbf{w}} = \mathbf{0}$, $\check{\Sigma} = \infty\mathbf{I}$, $\check{a} = -D/2$ and $\check{b} = 0$. The corresponding posterior is given by

$$p(\mathbf{w}, \sigma^2 | \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \hat{\mathbf{w}}, \hat{\Sigma}, \hat{a}, \hat{b}) \quad (15.67)$$

$$\hat{\mathbf{w}} = \hat{\mathbf{w}}_{\text{mle}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (15.68)$$

$$\hat{\Sigma} = (\mathbf{X}^\top \mathbf{X})^{-1} \triangleq \mathbf{C} \quad (15.69)$$

$$\hat{a} = \frac{\nu}{2} \quad (15.70)$$

$$\hat{b} = \frac{s^2 \nu}{2} \quad (15.71)$$

$$s^2 \triangleq \frac{\|\mathbf{y} - \hat{\mathbf{y}}\|^2}{\nu} \quad (15.72)$$

$$\nu = N - D \quad (15.73)$$

Hence the posterior distribution of the weights is given by

$$p(\mathbf{w}|\mathcal{D}) = \mathcal{T}(\mathbf{w}|\hat{\mathbf{w}}, s^2 \mathbf{C}, \nu) \quad (15.74)$$

where $\hat{\mathbf{w}}$ is the MLE. The marginals for each weight therefore have the form

$$p(w_d|\mathcal{D}) = \mathcal{T}(w_d|\hat{w}_d, s^2 C_{dd}, \nu) \quad (15.75)$$

15.2.3.2 Connection to frequentist statistics

Interestingly, the posterior when using Jeffreys prior is formally equivalent to the **frequentist sampling distribution** of the MLE, which has the form

$$p(\hat{w}_d|\mathcal{D}^*) = \mathcal{T}(\hat{w}_d|w_d, s^2 C_{dd}, \nu) \quad (15.76)$$

where $\mathcal{D}^* = (\mathbf{X}, \mathbf{y}^*)$ is hypothetical data generated from the true model given the fixed inputs \mathbf{X} . In books on frequentist statistics, this is more commonly written in the following equivalent way (see e.g., [Ric95, p542]):

$$\frac{\hat{w}_d - w_d}{s\sqrt{C_{dd}}} \sim t_{N-D} \quad (15.77)$$

The sampling distribution is numerically the same as the posterior distribution in Equation (15.75) because $\mathcal{T}(w|\mu, \sigma^2, \nu) = \mathcal{T}(\mu|w, \sigma^2, \nu)$. However, it is semantically quite different, since the sampling distribution does not condition on the observed data, but instead is based on hypothetical data drawn from the model. See [BT73, p117] for more discussion of the equivalences between Bayesian and frequentist analysis of simple linear models when using uninformative priors.

15.2.3.3 Zellner's *g*-prior

It is often reasonable to assume an uninformative prior on σ^2 , since that is just a scalar that does not have much influence on the results, but using an uninformative prior for \mathbf{w} can be dangerous, since the strength of the prior controls how well regularized the model is, as we know from ridge regression.

A common compromise is to use an NIG prior with $\check{a} = -D/2$, $\check{b} = 0$ (to ensure $p(\sigma^2) \propto 1$) and $\check{\mathbf{w}} = \mathbf{0}$ and $\check{\Sigma} = g(\mathbf{X}^\top \mathbf{X})^{-1}$, where $g > 0$ plays a role analogous to $1/\lambda$ in ridge regression. This is called Zellner's **g-prior** [Zel86].¹ We see that the prior covariance is proportional to $(\mathbf{X}^\top \mathbf{X})^{-1}$ rather than \mathbf{I} ; this ensures that the posterior is invariant to scaling of the inputs, e.g., due to a change in the units of measurement [Min00a].

1. Note this prior is conditioned on the inputs \mathbf{X} , but not the outputs \mathbf{y} ; this is totally valid in a conditional (discriminative) model, where all calculations are conditioned on \mathbf{X} , which is treated like a fixed constant input.

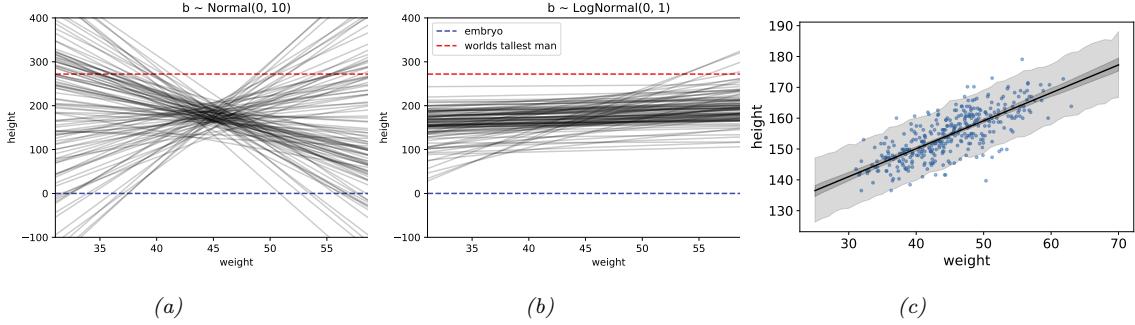


Figure 15.1: Linear regression for predicting height given weight, $y \sim \mathcal{N}(\alpha + \beta x, \sigma^2)$. (a) Prior predictive samples using a Gaussian prior for β . (b) Prior predictive samples using a log-Gaussian prior for β . (c) Posterior predictive samples using the log-Gaussian prior. The inner shaded band is the 95% credible interval for μ , representing epistemic uncertainty. The outer shaded band is the 95% credible interval for the observations y , which also adds data uncertainty due to σ . Adapted from Figures 4.5 and 4.10 of [McE20]. Generated by [linreg_height_weight.ipynb](#).

With this prior, the posterior becomes

$$p(\mathbf{w}, \sigma^2 | g, \mathcal{D}) = \text{NIG}(\mathbf{w}, \sigma^2 | \mathbf{w}_N, \mathbf{V}_N, a_N, b_N) \quad (15.78)$$

$$\mathbf{V}_N = \frac{g}{g+1} (\mathbf{X}^T \mathbf{X})^{-1} \quad (15.79)$$

$$\mathbf{w}_N = \frac{g}{g+1} \hat{\mathbf{w}}_{mle} \quad (15.80)$$

$$a_N = N/2 \quad (15.81)$$

$$b_N = \frac{s^2}{2} + \frac{1}{2(g+1)} \hat{\mathbf{w}}_{mle}^T \mathbf{X}^T \mathbf{X} \hat{\mathbf{w}}_{mle} \quad (15.82)$$

Various approaches have been proposed for setting g , including cross validation, empirical Bayes [Min00a; GF00], hierarchical Bayes [Lia+08], etc.

15.2.4 Informative priors

In many problems, it is possible to use domain knowledge to come up with plausible priors. As an example, we consider the problem of predicting the height of a person given their weight. We will use a dataset collected from Kalahari foragers by the anthropologist Nancy Howell (this example is from [McE20, p93]).

Let x_i be the weight (in kg) and y_i be height (in cm) of the i 'th person, and let \bar{x} be the mean of the inputs. The observation model is given by

$$y_i \sim \mathcal{N}(\mu_i, \sigma) \quad (15.83)$$

$$\mu_i = \alpha + \beta(x_i - \bar{x}) \quad (15.84)$$

We see that the intercept α is the predicted output if $x_i = \bar{x}$, and the slope β is the predicted change in height per unit change in weight above or below the average weight.

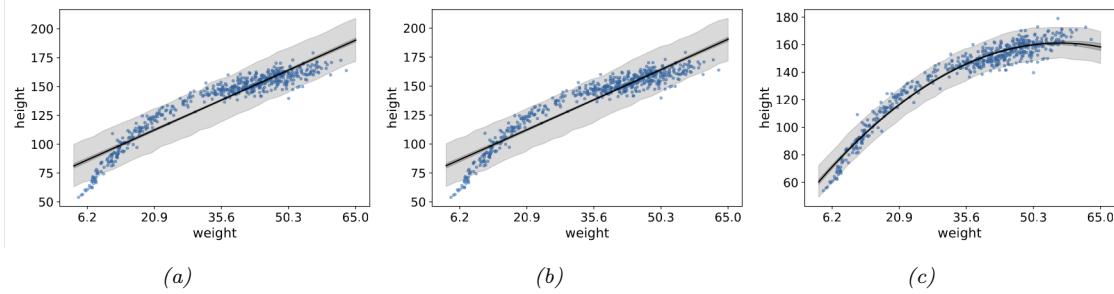


Figure 15.2: Linear regression for predicting height given weight for the full dataset (including children) using polynomial regression. (a) Posterior fit for linear model with log-Gaussian prior for β_1 . (b) Posterior fit for quadratic model with log-Gaussian prior for β_2 . (c) Posterior fit for quadratic model with Gaussian prior for β_2 . Adapted from Figure 4.11 of [McE20]. Generated by [linreg_height_weight.ipynb](#).

The question is: what priors should we use? To be truly Bayesian, we should set these before looking at the data. A sensible prior for α is the height of a “typical person”, with some spread. We use $\alpha \sim \mathcal{N}(178, 20)$, since the author of the book from which this example is taken is 178cm. By using a standard deviation of 20, the prior puts 95% probability on the broad range of 178 ± 40 .

What about the prior for β ? It is tempting to use a **vague prior**, or **weak prior**, such as $\beta \sim \mathcal{N}(0, 10)$, which is similar to a flat (uniform) prior, but more concentrated at 0 (a form of mild regularization). To see if this is reasonable, we can compute samples from the **prior predictive distribution**, i.e., we sample $(\alpha_s, \beta_s) \sim p(\alpha)p(\beta)$, and then plot $\alpha_s x + \beta_s$ for a range of x values, for different samples $s = 1 : S$. The results are shown in Figure 15.1a. We see that this is not a very sensible prior. For example, we see that it suggests that it is just as likely for the height to decrease with weight as increase with weight, which is not plausible. In addition, it predicts heights which are larger than the world’s tallest person (272 cm) and smaller than the world’s shortest person (an embryo, of size 0).

We can encode the monotonically increasing relationship between weight and height by restricting β to be positive. An easy way to do this is to use a log-normal or log-Gaussian prior. (If $\tilde{\beta} = \log(\beta)$ is Gaussian, then $e^{\tilde{\beta}}$ must be positive.) Specifically, we will assume $\beta \sim \mathcal{LN}(0, 1)$. Samples from this prior are shown in Figure 15.1b. This is much more reasonable.

Finally we must choose a prior over σ . In [McE20] they use $\sigma \sim \text{Unif}(0, 50)$. This ensures that σ is positive, and that the prior predictive distribution for the output is within 100cm of the average height. However, it is usually easier to specify the expected value for σ than an upper bound. To do this, we can use $\sigma \sim \text{Expon}(\lambda)$, where λ is the rate. We then set $\mathbb{E}[\sigma] = 1/\lambda$ to the value of the standard deviation that we expect. For example, we can use the empirical standard deviation of the data.

Since these priors are no longer conjugate, we cannot compute the posterior in closed form. However, we can use a variety of approximate inference methods. In this simple example, it suffices to use a quadratic (Laplace) approximation (see Section 7.4.3). The results are shown in Figure 15.1c, and look sensible.

So far, we have only considered a subset of the data, corresponding to adults over the age of 18. If we include children, we find that the mapping from weight to height is nonlinear. This is illustrated

in Figure 15.2a. We can fix this problem by using **polynomial regression**. For example, consider a quadratic expansion of the standardized features x_i :

$$\mu_i = \alpha + \beta_1 x_i + \beta_2 x_i^2 \quad (15.85)$$

If we use a log-Gaussian prior for β_2 , we find that the model is too constrained, and it underfits. This is illustrated in Figure 15.2b. The reason is that we need to use an inverted quadratic with a negative coefficient, but since this is disallowed by the prior, the model ends up not using this degree of freedom (we find $\mathbb{E}[\beta_2|\mathcal{D}] \approx 0.08$). If we use a Gaussian prior on β_2 , we avoid this problem, illustrated in Figure 15.2c.

This example shows that it can be useful to think about the functional form of the mapping from inputs to outputs in order to specify sensible priors.

15.2.5 Spike and slab prior

It is often useful to be able to select a subset of the input features when performing prediction, either to reduce overfitting, or to improve interpretability of the model. This can be achieved if we ensure that the weight vector \mathbf{w} is **sparse** (i.e., has many zero elements), since if $w_d = 0$, then x_d plays no role in the inner product $\mathbf{w}^\top \mathbf{x}$.

The canonical way to achieve sparsity when using Bayesian inference is to use a **spike-and-slab** (**SS**) prior [MB88], which has the form of a 2 component mixture model, with one component being a “spike” at 0, and the other being a uniform “slab” between $-a$ and a :

$$p(\mathbf{w}) = \prod_{d=1}^D (1 - \pi) \delta(w_d) + \pi \text{Unif}(w_d | -a, a) \quad (15.86)$$

where π is the prior probability that each coefficient is non-zero. The corresponding log prior on the coefficients is thus

$$\log p(\mathbf{w}) = \|\mathbf{w}\|_0 \log(1 - \pi) + (D - \|\mathbf{w}\|_0) \log \pi = -\lambda \|\mathbf{w}\|_0 + \text{const} \quad (15.87)$$

where $\lambda = \log \frac{\pi}{1-\pi}$ controls the sparsity of the model, and $\|\mathbf{w}\|_0 = \sum_{d=1}^D \mathbb{I}(w_d \neq 0)$ is the ℓ_0 **norm** of the weights. Thus MAP estimation with a spike and slab prior is equivalent ℓ_0 **regularization**; this penalizes the number of non-zero coefficients. Interestingly, posterior samples will also be sparse.

By contrast, consider using a Laplace prior. The **lasso** estimator uses MAP estimation, which results in a sparse estimate. However, posterior samples are not sparse. Interestingly, [EY09] show theoretically (and [SPZ09] confirm experimentally) that using the posterior mean with a spike-and-slab prior also results in better prediction accuracy than using the posterior mode with a Laplace prior.

In practice, we often approximate the uniform slab with a broad Gaussian distribution,

$$p(\mathbf{w}) = \prod_d (1 - \pi) \delta(w_d) + \pi \mathcal{N}(w_d | 0, \sigma_w^2) \quad (15.88)$$

As $\sigma_w^2 \rightarrow \infty$, the second term approaches a uniform distribution over $[-\infty, +\infty]$. We can implement the mixture model by associating a binary random variable, $s_d \sim \text{Ber}(\pi)$, with each coefficient, to indicate if the coefficient is “on” or “off”.

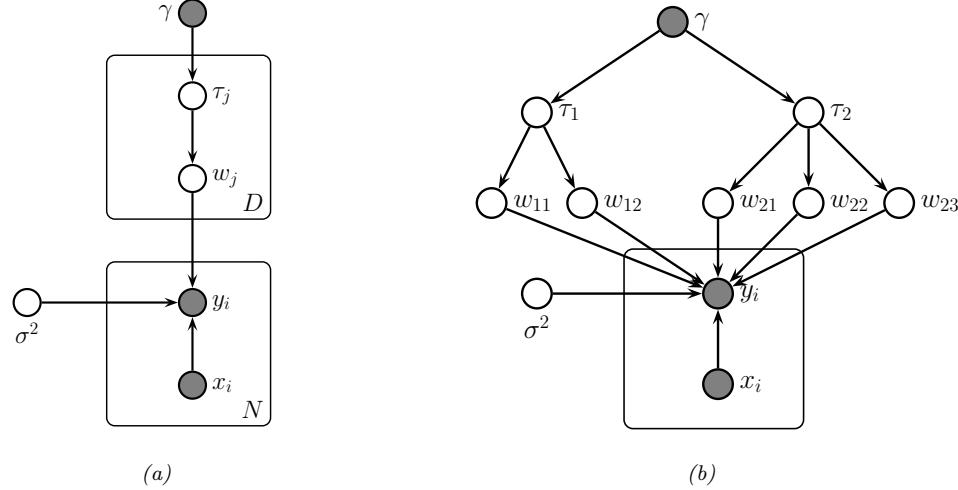


Figure 15.3: (a) Representing lasso using a Gaussian scale mixture prior. (b) Graphical model for group lasso with 2 groups, the first has size $G_1 = 2$, the second has size $G_2 = 3$.

Unfortunately, MAP estimation (not to mention full Bayesian inference) with such discrete mixture priors is computationally difficult. Various approximate inference methods have been proposed, including greedy search (see e.g., [SPZ09]) or MCMC (see e.g., [HS09]).

15.2.6 Laplace prior (Bayesian lasso)

A computationally cheap way to achieve sparsity is to perform MAP estimation with a Laplace prior by minimizing the penalized negative log likelihood:

$$\text{PNLL}(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) - \log p(\mathbf{w}|\lambda) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1 \quad (15.89)$$

where $\|\mathbf{w}\|_1 \triangleq \sum_{d=1}^D |w_d|$ is the ℓ_1 norm of \mathbf{w} . This method is called **lasso**, which stands for “least absolute shrinkage and selection operator” [Tib96]. See Section 11.4 of the prequel to this book, [Mur22], for details.

In this section, we discuss posterior inference with this prior; this is known as the **Bayesian lasso** [PC08]. In particular, we assume the following prior:

$$p(\mathbf{w}|\sigma^2) = \prod_j \frac{\lambda}{2\sqrt{\sigma^2}} e^{-\lambda|w_j|/\sqrt{\sigma^2}} \quad (15.90)$$

(Note that conditioning the prior on σ^2 is important to ensure that the full posterior is unimodal.)

To simplify inference, we will represent the Laplace prior as a Gaussian scale mixture, which we discussed in Section 28.2.3.2. In particular, one can show that the Laplace distribution is an infinite

weighted sum of Gaussians, where the precision comes from a gamma distribution:

$$\text{Laplace}(w|0, \lambda) = \int \mathcal{N}(w|0, \tau^2) \text{Ga}(\tau^2|1, \frac{\lambda^2}{2}) d\tau^2 \quad (15.91)$$

We can therefore represent the Bayesian lasso model as a hierarchical latent variable model, as shown in Figure 15.3a. The corresponding joint distribution has the following form:

$$p(\mathbf{y}, \mathbf{w}, \boldsymbol{\tau}, \sigma^2 | \mathbf{X}) = \mathcal{N}(\mathbf{y} | \mathbf{X}\mathbf{w}, \sigma^2 \mathbf{I}_N) \left[\prod_j \mathcal{N}(w_j | 0, \sigma^2 \tau_j^2) \text{Ga}(\tau_j^2 | 1, \lambda^2/2) \right] p(\sigma^2) \quad (15.92)$$

We can also create a GSM to match the **group lasso** prior, which sets multiple coefficients to zero at the same time:

$$\mathbf{w}_g | \sigma^2, \tau_g^2 \sim \mathcal{N}(\mathbf{0}, \sigma^2 \tau_g^2 \mathbf{I}_{d_g}) \quad (15.93)$$

$$\tau_g^2 \sim \text{Ga}(\frac{d_g + 1}{2}, \frac{\lambda^2}{2}) \quad (15.94)$$

where d_g is the size of group g . So we see that there is one variance term per group, each of which comes from a gamma prior, whose shape parameter depends on the group size, and whose rate parameter is controlled by γ .

Figure 15.3b gives an example, where we have 2 groups, one of size 2 and one of size 3. This picture makes it clearer why there should be a grouping effect. For example, suppose $w_{1,1}$ is small; then τ_1^2 will be estimated to be small, which will force $w_{1,2}$ to be small, due to shrinkage (cf. Section 3.6). Conversely, suppose $w_{1,1}$ is large; then τ_1^2 will be estimated to be large, which will allow $w_{1,2}$ to be become large as well.

Given these hierarchical models, we can easily derive a Gibbs sampling algorithm (Section 12.3) to sample from the posterior (see e.g., [PC08]). Unfortunately, these posterior samples are not sparse, even though the MAP estimate is sparse. This is because the prior puts infinitesimal probability on the event that each coefficient is zero.

15.2.7 Horseshoe prior

The Laplace prior is not suitable for sparse Bayesian models, because posterior samples are not sparse. The spike and slab prior does not have this problem but is often too slow to use (although see [BRG20]). Fortunately, it is possible to devise continuous priors (without discrete latent variables) that are both sparse and computationally efficient. One popular prior of this type is the **horseshoe prior** [CPS10], so-named because of the shape of its density function.

In the horseshoe prior, instead of using a Laplace prior for each weight, we use the following Gaussian scale mixture:

$$w_j \sim \mathcal{N}(0, \lambda_j^2 \tau^2) \quad (15.95)$$

$$\lambda_j \sim \mathcal{C}_+(0, 1) \quad (15.96)$$

$$\tau^2 \sim \mathcal{C}_+(0, 1) \quad (15.97)$$

where $\mathcal{C}_+(0, 1)$ is the half-Cauchy distribution (Section 2.2.4), λ_j is a local shrinkage factor, and τ^2 is a global shrinkage factor. The Cauchy distribution has very fat tails, so λ_j is likely to be either 0 or very far from 0, which emulates the spike and slab prior, but in a continuous way. For more details, see e.g., [Bha+19].

15.2.8 Automatic relevancy determination

An alternative to using posterior inference with a sparsity promoting prior is to use posterior inference with a Gaussian prior, $w_j \sim \mathcal{N}(0, 1/\alpha_j)$, but where we use empirical Bayes to optimize the precisions α_j . That is, we first compute $\hat{\boldsymbol{\alpha}} = \text{argmax}_{\boldsymbol{\alpha}} p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha})$, and then compute $\hat{\mathbf{w}} = \text{argmax}_{\mathbf{w}} \mathcal{N}(\mathbf{w}|\mathbf{0}, \hat{\boldsymbol{\alpha}}^{-1})$. Perhaps surprisingly, we will see that this results in a sparse estimate, for reasons we explain in Section 15.2.8.2.

This technique is known as **sparse Bayesian learning** [Tip01] or **automatic relevancy determination (ARD)** [Mac95; Nea96]. It has also been called **NUV** estimation, which stands for “normal prior with unknown variance” [Loe+16]. It was originally developed for neural networks (where sparsity is applied to the first layer weights), but here we apply it to linear models.

15.2.8.1 ARD for linear models

In this section, we explain ARD in more detail, by applying it to linear regression. The likelihood is $p(\mathbf{y}|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(\mathbf{y}|\mathbf{w}^\top \mathbf{x}, 1/\beta)$, where $\beta = 1/\sigma^2$. The prior is $p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{A}^{-1})$, where $\mathbf{A} = \text{diag}(\boldsymbol{\alpha})$. The marginal likelihood can be computed analytically (using Equation (2.129)) as follows:

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, (1/\beta)\mathbf{I}_N) \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{A}^{-1}) d\mathbf{w} \quad (15.98)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top) \quad (15.99)$$

$$= \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_{\boldsymbol{\alpha}}) \quad (15.100)$$

where $\mathbf{C}_{\boldsymbol{\alpha}} \triangleq \beta^{-1}\mathbf{I}_N + \mathbf{X}\mathbf{A}^{-1}\mathbf{X}^\top$. This is very similar to the marginal likelihood under the spike-and-slab prior (Section 15.2.5), which is given by

$$p(\mathbf{y}|\mathbf{X}, \mathbf{s}, \sigma_w^2, \sigma_y^2) = \int \mathcal{N}(\mathbf{y}|\mathbf{X}_{\mathbf{s}}\mathbf{w}_{\mathbf{s}}, \sigma_y^2\mathbf{I}) \mathcal{N}(\mathbf{w}_{\mathbf{s}}|\mathbf{0}_{\mathbf{s}}, \sigma_w^2\mathbf{I}) d\mathbf{w}_{\mathbf{s}} = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_{\mathbf{s}}) \quad (15.101)$$

where $\mathbf{C}_{\mathbf{s}} = \sigma_y^2\mathbf{I}_N + \sigma_w^2\mathbf{X}_{\mathbf{s}}\mathbf{X}_{\mathbf{s}}^\top$. (Here $\mathbf{X}_{\mathbf{s}}$ refers to the design matrix where we select only the columns of \mathbf{X} where $s_d = 1$.) The difference is that we have replaced the binary $s_j \in \{0, 1\}$ variables with continuous $\alpha_j \in \mathbb{R}^+$, which makes the optimization problem easier.

The objective is the log marginal likelihood, given by

$$\ell(\boldsymbol{\alpha}, \beta) = -2 \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \log |\mathbf{C}_{\boldsymbol{\alpha}}| + \mathbf{y}^\top \mathbf{C}_{\boldsymbol{\alpha}}^{-1} \mathbf{y} \quad (15.102)$$

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$, some of which we discuss in Section 15.2.8.3.

ARD can be used as an alternative to ℓ_1 regularization. Although the ARD objective is not convex, it tends to give much sparser results [WW12]. In addition, it can be shown [WRN10] that the ARD objective has many fewer local optima than the ℓ_0 -regularized objective, and hence is much easier to optimize.

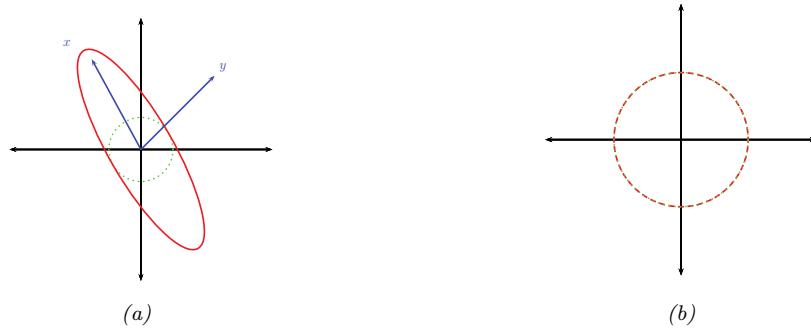


Figure 15.4: Illustration of why ARD results in sparsity. The vector of inputs \mathbf{x} does not point towards the vector of outputs \mathbf{y} , so the feature should be removed. (a) For finite α , the probability density is spread in directions away from \mathbf{y} . (b) When $\alpha = \infty$, the probability density at \mathbf{y} is maximized. Adapted from Figure 8 of [Tip01].

15.2.8.2 Why does ARD result in a sparse solution?

Once we have estimated $\boldsymbol{\alpha}$ and β , we can compute the posterior over the parameters using Bayes' rule for Gaussians, to get $p(\mathbf{w}|\mathcal{D}, \hat{\boldsymbol{\alpha}}, \hat{\beta}) = \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \hat{\Sigma})$, where $\hat{\Sigma}^{-1} = \hat{\beta}\mathbf{X}^T\mathbf{X} + \mathbf{A}$ and $\hat{\mathbf{w}} = \hat{\beta}\hat{\Sigma}\mathbf{X}^T\mathbf{y}$. If we have $\hat{\alpha}_d \approx \infty$, then $\hat{w}_d \approx 0$, so the solution vector will be sparse.

We now give an intuitive argument, based on [Tip01], about when such a sparse solution may be optimal. We shall assume $\beta = 1/\sigma^2$ is fixed for simplicity. Consider a 1d linear regression with 2 training examples, so $\mathbf{X} = \mathbf{x} = (x_1, x_2)$, and $\mathbf{y} = (y_1, y_2)$. We can plot \mathbf{x} and \mathbf{y} as vectors in the plane, as shown in Figure 15.4. Suppose the feature is irrelevant for predicting the response, so \mathbf{x} points in a nearly orthogonal direction to \mathbf{y} . Let us see what happens to the marginal likelihood as we change α . The marginal likelihood is given by $p(\mathbf{y}|\mathbf{x}, \alpha, \beta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\alpha)$, where $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I} + \frac{1}{\alpha}\mathbf{x}\mathbf{x}^T$. If α is finite, the posterior will be elongated along the direction of \mathbf{x} , as in Figure 15.4(a). However, if $\alpha = \infty$, we have $\mathbf{C}_\alpha = \frac{1}{\beta}\mathbf{I}$, which is spherical, as in Figure 15.4(b). If $|\mathbf{C}_\alpha|$ is held constant, the latter assigns higher probability density to the observed response vector \mathbf{y} , so this is the preferred solution. In other words, the marginal likelihood “punishes” solutions where α_d is small but $\mathbf{X}_{:,d}$ is irrelevant, since these waste probability mass. It is more parsimonious (from the point of view of Bayesian Occam's razor) to eliminate redundant dimensions.

Another way to understand the sparsity properties of ARD is as approximate inference in a hierarchical Bayesian model [BT00]. In particular, suppose we put a conjugate prior on each precision, $\alpha_d \sim \text{Ga}(a, b)$, and on the observation precision, $\beta \sim \text{Ga}(c, d)$. Since exact inference with a Student prior is intractable, we can use variational Bayes (Section 10.3.3), with a factored posterior approximation of the form

$$q(\mathbf{w}, \boldsymbol{\alpha}) = q(\mathbf{w})q(\boldsymbol{\alpha}) \approx \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \prod_d \text{Ga}(\alpha_d | \hat{a}_d, \hat{b}_d) \quad (15.103)$$

ARD approximates $q(\boldsymbol{\alpha})$ by a point estimate. However, in VB, we integrate out $\boldsymbol{\alpha}$; the resulting

posterior marginal $q(\mathbf{w})$ on the weights is given by

$$p(\mathbf{w}|\mathcal{D}) = \int \mathcal{N}(\mathbf{w}|\mathbf{0}, \text{diag}(\boldsymbol{\alpha})^{-1}) \prod_d \text{Ga}(\alpha_d | \hat{a}_d, \hat{b}) d\boldsymbol{\alpha} \quad (15.104)$$

This is a Gaussian scale mixture, and can be shown to be the same as a multivariate Student distribution (see Section 28.2.3.1), with non-diagonal covariance. Note that the Student has a large spike at 0, which intuitively explains why the posterior mean (which, for a Student distribution, is equal to the posterior mode) is sparse.

Finally, we can also view ARD as a MAP estimation problem with a **non-factorial prior** [WN07]. Intuitively, the dependence between the w_j parameters arises, despite the use of a diagonal Gaussian prior, because the prior precision α_j is estimated based after marginalizing out all \mathbf{w} , and hence depends on all the features. Interestingly, [WRN10] prove that MAP estimation with non-factorial priors is strictly better than MAP estimation with any possible factorial prior in the following sense: the non-factorial objective always has fewer local minima than factorial objectives, while still satisfying the property that the global optimum of the non-factorial objective corresponds to the global optimum of the ℓ_0 objective — a property that ℓ_1 regularization, which has no local minima, does not enjoy.

15.2.8.3 Algorithms for ARD

There are various algorithms for optimizing $\ell(\boldsymbol{\alpha}, \beta)$. One approach is to use EM, in which we compute $p(\mathbf{w}|\mathcal{D}, \boldsymbol{\alpha})$ in the E step and then maximize $\boldsymbol{\alpha}$ in the M step. In variational Bayes, we infer both \mathbf{w} and $\boldsymbol{\alpha}$ (see [Dru08] for details). In [WN10], they present a method based on iteratively reweighted ℓ_1 estimation.

Recently, [HXW17] showed that the nested iterative computations performed these methods can be emulated by a recurrent neural network (Section 16.3.4). Furthermore, by training this model, it is possible to achieve much faster convergence than manually designed optimization algorithms.

15.2.8.4 Relevance vector machines

Suppose we create a linear regression model of the form $p(y|\mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^\top \phi(\mathbf{x}), \sigma^2)$, where $\phi(\mathbf{x}) = [\mathcal{K}(\mathbf{x}, \mathbf{x}_1), \dots, \mathcal{K}(\mathbf{x}, \mathbf{x}_N)]$, where $\mathcal{K}()$ is a kernel function (Section 18.2) and $\mathbf{x}_1, \dots, \mathbf{x}_N$ are the N training points. This is called **kernel basis function expansion**, and transforms the input from $\mathbf{x} \in \mathcal{X}$ to $\phi(\mathbf{x}) \in \mathbb{R}^N$. Obviously this model has $O(N)$ parameters, and hence is nonparametric. However, we can use ARD to select a small subset of the exemplars. This technique is called the relevance vector machine (RVM) [Tip01; TF03].

15.2.9 Multivariate linear regression

This section is written by Xinglong Li.

In this section, we consider the **multivariate linear regression** model, which has the form

$$\mathbf{Y} = \mathbf{WX} + \mathbf{E} \quad (15.105)$$

where $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$ is the matrix of regression coefficient, $\mathbf{X} \in \mathbb{R}^{N_x \times N}$ is the matrix of input features (with each row being an input variable and each column being an observation), $\mathbf{Y} \in \mathbb{R}^{N_y \times N}$ is the

matrix of responses (with each row being an output variable and each *column* being an observation), and $\mathbf{E} = [\mathbf{e}_1, \dots, \mathbf{e}_N]$ is the matrix of residual errors, where $\mathbf{e}_i \stackrel{iid}{\sim} \mathcal{N}(\mathbf{0}, \Sigma)$. It can be seen from the definition that given Σ , \mathbf{W} and \mathbf{X} , columns of \mathbf{Y} are independently random variables following multivariate normal distributions. So the likelihood of the observation is

$$p(\mathbf{Y}|\mathbf{W}, \mathbf{X}, \Sigma) = \frac{1}{(2\pi)^{N_y \times N} |\Sigma|^{N/2}} \exp \left(\sum_{i=1}^N -\frac{1}{2} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i)^T \Sigma^{-1} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i) \right) \quad (15.106)$$

$$= \frac{1}{(2\pi)^{N_y \times N} |\Sigma|^{N/2}} \exp \left(-\frac{1}{2} \text{tr} ((\mathbf{Y} - \mathbf{W}\mathbf{X})^T \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X})) \right) \quad (15.107)$$

$$= \mathcal{MN}(\mathbf{Y}|\mathbf{W}\mathbf{X}, \Sigma, \mathbf{I}_{N \times N}), \quad (15.108)$$

The conjugate prior for this is the **matrix normal inverse Wishart** distribution,

$$\mathbf{W}, \Sigma \sim \text{MNIW}(\mathbf{M}_0, \mathbf{V}_0, \nu_0, \Psi_0) \quad (15.109)$$

where the MNIW is defined by

$$\mathbf{W}|\Sigma \sim \mathcal{MN}(\mathbf{M}_0, \Sigma_0, \mathbf{V}_0) \quad (15.110)$$

$$\Sigma \sim \text{IW}(\nu_0, \Psi_0), \quad (15.111)$$

where $\mathbf{V}_0 \in \mathbb{R}_{++}^{N_x \times N_x}$, $\Psi_0 \in \mathbb{R}_{++}^{N_y \times N_y}$ and $\nu_0 > N_x - 1$ is the degree of freedom of the inverse Wishart distribution.

The posterior distribution of $\{\mathbf{W}, \Sigma\}$ still follows a matrix normal inverse Wishart distribution. We follow the derivation in [Fox09, App.F]. The density of the joint distribution is

$$p(\mathbf{Y}, \mathbf{W}, \Sigma) \propto |\Sigma|^{-(\nu_0 + N_y + 1 + N_x + N)/2} \times \exp \left\{ -\frac{1}{2} \text{tr}(\Omega_0) \right\} \quad (15.112)$$

$$\Omega_0 \triangleq \Psi_0 \Sigma^{-1} + (\mathbf{Y} - \mathbf{W}\mathbf{X})^T \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^T \Sigma^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0 \quad (15.113)$$

We first aggregate items including \mathbf{W} in the exponent so that it takes the form of a matrix normal distribution. This is similar to the ‘‘completing the square’’ technique that we used in deriving the conjugate posterior for multivariate normal distributions in Section 3.4.4.3. Specifically,

$$\text{tr}[(\mathbf{Y} - \mathbf{W}\mathbf{X})^T \Sigma^{-1} (\mathbf{Y} - \mathbf{W}\mathbf{X}) + (\mathbf{W} - \mathbf{M}_0)^T \Sigma^{-1} (\mathbf{W} - \mathbf{M}_0) \mathbf{V}_0] \quad (15.114)$$

$$= \text{tr}(\Sigma^{-1}[(\mathbf{Y} - \mathbf{W}\mathbf{X})(\mathbf{Y} - \mathbf{W}\mathbf{X})^T + (\mathbf{W} - \mathbf{M}_0)\mathbf{V}_0(\mathbf{W} - \mathbf{M}_0)^T]) \quad (15.115)$$

$$= \text{tr}(\Sigma^{-1}[\mathbf{W}\mathbf{S}_{xx}\mathbf{W}^T - 2\mathbf{S}_{yx}\mathbf{W}^T + \mathbf{S}_{yy}]) \quad (15.116)$$

$$= \text{tr}(\Sigma^{-1}[(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})\mathbf{S}_{xx}(\mathbf{W} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1})^T + \mathbf{S}_{y|x}]). \quad (15.117)$$

where

$$\mathbf{S}_{xx} = \mathbf{X}\mathbf{X}^T + \mathbf{V}_0, \quad \mathbf{S}_{yx} = \mathbf{Y}\mathbf{X}^T + \mathbf{M}_0\mathbf{V}_0, \quad (15.118)$$

$$\mathbf{S}_{yy} = \mathbf{Y}\mathbf{Y}^T + \mathbf{M}_0\mathbf{V}_0\mathbf{M}_0^T, \quad \mathbf{S}_{y|x} = \mathbf{S}_{yy} - \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}\mathbf{S}_{yx}^T. \quad (15.119)$$

Therefore, it can be seen from Equation (15.117) that given Σ , \mathbf{W} follows a matrix normal distribution

$$\mathbf{W}|\Sigma, \mathbf{X}, \mathbf{Y} \sim \mathcal{MN}(\mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}, \Sigma, \mathbf{S}_{xx}). \quad (15.120)$$

Marginalizing out \mathbf{W} (which corresponds to removing the terms including \mathbf{W} in the exponent in Equation (15.113)), it can be shown that the posterior distribution of Σ is an inverse Wishart distribution. In fact, by replacing Equation (15.117) to the corresponding terms in Equation (15.113), it can be seen that the only terms left after integrating out \mathbf{W} are $\Sigma^{-1}\Psi$ and $\Sigma^{-1}\mathbf{S}_{y|x}$, which indicates that the scale matrix of the posterior inverse Wishart distribution is $\Psi_0 + \mathbf{S}_{y|x}$.

In conclusion, the joint posterior distribution of $\{\mathbf{W}, \Sigma\}$ given the observation is

$$\mathbf{W}, \Sigma | \mathbf{X}, \mathbf{Y} \sim \text{MNIW}(\mathbf{M}_1, \mathbf{V}_1, \nu_1, \Psi_1) \quad (15.121)$$

$$\mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \quad (15.122)$$

$$\mathbf{V}_1 = \mathbf{S}_{xx} \quad (15.123)$$

$$\nu_1 = N + \nu_0 \quad (15.124)$$

$$\Psi_1 = \Psi_0 + \mathbf{S}_{y|x} \quad (15.125)$$

The MAP estimate of \mathbf{W} and Σ are the mode of the posterior matrix normal inverse Wishart distribution. To derive this, notice that \mathbf{W} only appears in the matrix normal density function in the posterior, so the matrix \mathbf{W} maximizing the posterior density of $\{\mathbf{W}, \Sigma\}$ is the matrix \mathbf{W} that maximizes the matrix normal posterior of \mathbf{W} . So the MAP estimate of \mathbf{W} is $\hat{\mathbf{W}} = \mathbf{M}_1 = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1}$, and this holds for any value of Σ . By plugging $\mathbf{W} = \hat{\mathbf{W}}$ into the joint posterior of $\{\mathbf{W}, \Sigma\}$, and taking derivatives over Σ , it can be seen that the matrix maximizing the density is $(\nu_1 + N_y + N_x + 1)^{-1}\Psi_1$. Since Ψ_1 is positive definite, it is the MAP estimate of Σ .

In conclusion, the MAP estimate of $\{\mathbf{W}, \Sigma\}$ are

$$\hat{\mathbf{W}} = \mathbf{S}_{yx}\mathbf{S}_{xx}^{-1} \quad (15.126)$$

$$\hat{\Sigma} = \frac{1}{\nu_1 + N_y + N_x + 1} (\Psi_0 + \mathbf{S}_{y|x}) \quad (15.127)$$

15.3 Logistic regression

Logistic regression is a very widely used discriminative classification model that maps input vectors $\mathbf{x} \in \mathbb{R}^D$ to a distribution over class labels, $y \in \{1, \dots, C\}$. If $C = 2$, this is known as **binary logistic regression**, and if $C > 2$, it is known as **multinomial logistic regression**, or alternatively, **multiclass logistic regression**.

15.3.1 Binary logistic regression

In the binary case, where $y \in \{0, 1\}$, the model has the following form

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = \text{Ber}(y|\sigma(\mathbf{w}^\top \mathbf{x} + b)) \quad (15.128)$$

where \mathbf{w} are the weights, b is the bias (offset), and σ is the **sigmoid** or **logistic** function, defined by

$$\sigma(a) \triangleq \frac{1}{1 + e^{-a}} \quad (15.129)$$

Let $\eta_n = \mathbf{w}^\top \mathbf{x}_n + b$ be the **logits** for example n , and $\mu_n = \sigma(\eta_n) = p(y = 1 | \mathbf{x}_n)$ be the mean of the output. Then we can write the log likelihood as the negative cross entropy:

$$\log p(\mathcal{D} | \boldsymbol{\theta}) = \log \prod_{n=1}^N \mu_n^{y_n} (1 - \mu_n)^{1-y_n} = \sum_{n=1}^N y_n \log \mu_n + (1 - y_n) \log(1 - \mu_n) \quad (15.130)$$

We can expand this equation into a more explicit form (that is commonly seen in implementations) by performing some simple algebra. First note that

$$\mu_n = \frac{1}{1 + e^{-\eta_n}} = \frac{e^{\eta_n}}{1 + e^{\eta_n}}, \quad 1 - \mu_n = 1 - \frac{e^{\eta_n}}{1 + e^{\eta_n}} = \frac{1}{1 + e^{\eta_n}} \quad (15.131)$$

Hence

$$\log p(\mathcal{D} | \boldsymbol{\theta}) = \sum_{n=1}^N y_n [\log e^{\eta_n} - \log(1 + e^{\eta_n})] + (1 - y_n) [\log 1 - \log(1 + e^{\eta_n})] \quad (15.132)$$

$$= \sum_{n=1}^N y_n [\eta_n - \log(1 + e^{\eta_n})] + (1 - y_n) [-\log(1 + e^{\eta_n})] \quad (15.133)$$

$$= \sum_{n=1}^N y_n \eta_n - \sum_{n=1}^N \log(1 + e^{\eta_n}) \quad (15.134)$$

Note that the $\log(1 + e^a)$ function is often implemented using `np.log1p(np.exp(a))`.

15.3.2 Multinomial logistic regression

Multinomial logistic regression is a discriminative classification model of the following form:

$$p(y | \mathbf{x}; \boldsymbol{\theta}) = \text{Cat}(y | \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})) \quad (15.135)$$

where $\mathbf{x} \in \mathbb{R}^D$ is the input vector, $y \in \{1, \dots, C\}$ is the class label, \mathbf{W} is a $C \times D$ weight matrix, \mathbf{b} is C -dimensional bias vector, and `softmax()` is the **softmax function**, defined as

$$\text{softmax}(\mathbf{a}) \triangleq \left[\frac{e^{a_1}}{\sum_{c'=1}^C e^{a_{c'}}}, \dots, \frac{e^{a_C}}{\sum_{c'=1}^C e^{a_{c'}}} \right] \quad (15.136)$$

If we define the logits as $\boldsymbol{\eta}_n = \mathbf{W}\mathbf{x}_n + \mathbf{b}$, the probabilities as $\boldsymbol{\mu}_n = \text{softmax}(\boldsymbol{\eta}_n)$, and let \mathbf{y}_n be the one-hot encoding of the label y_n , then the log likelihood can be written as the negative cross entropy:

$$\log p(\mathcal{D} | \boldsymbol{\theta}) = \log \prod_{n=1}^N \prod_{c=1}^C \mu_{nc}^{y_{nc}} = \sum_{n=1}^N \sum_{c=1}^C y_{nc} \log \mu_{nc} \quad (15.137)$$

15.3.3 Dealing with class imbalance and the long tail

In many problems, some classes are much rarer than others; this problem is called **class imbalance**. In such a setting, standard maximum likelihood training may not work well, since it is designed to minimize (a bound on) the 0-1 loss, which can be dominated by the most frequent classes. A natural alternative is to consider the **balanced error rate**, which computes the average of the per-class error rates of classifier f :

$$\text{BER}(f) = \frac{1}{C} \sum_{y=1}^C p_{\mathbf{x}|y}^* \left(y \notin \operatorname{argmax}_{y' \in \mathcal{Y}} f_{y'}(\mathbf{x}) \right) \quad (15.138)$$

where p^* is the true distribution. The classifier that optimizes this loss, f^* , must satisfy

$$\operatorname{argmax}_{y \in \mathcal{Y}} f_y^*(\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} p_{\text{bal}}^*(y|\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} p^*(\mathbf{x}|y) \quad (15.139)$$

where $p_{\text{bal}}^*(y|\mathbf{x}) \propto \frac{1}{C} p^*(\mathbf{x}|y)$ is the predictor when using balanced classes. Thus to minimize the BER, we should use the class-conditional likelihood, $p(\mathbf{x}|y)$, rather than the class posterior, $p(y|\mathbf{x})$.

In [Men+21], they propose a simple scheme called **logit adjustment** that can achieve this optimal classifier. We assume the model computes logits using $f_y(\mathbf{x}) = \mathbf{w}_y^\top \phi(\mathbf{x})$, where $\phi(\mathbf{x}) = \mathbf{x}$ for a GLM. In the post-hoc version, the model is trained in the usual way, and then at prediction time, we use

$$\operatorname{argmax}_{y \in \mathcal{Y}} p(\mathbf{x}|y) = \operatorname{argmax}_{y \in \mathcal{Y}} \frac{p(y|\mathbf{x})p(\mathbf{x})}{p(y)} = \operatorname{argmax}_{y \in \mathcal{Y}} \frac{\exp(\mathbf{w}_y^\top \phi(\mathbf{x}))}{\pi_y} \quad (15.140)$$

where $\pi_y = p(y)$ is the empirical label prior. In practice, it is helpful to introduce a tuning parameter $\tau > 0$ and to use the predictor

$$\hat{f}(\mathbf{x}) = \operatorname{argmax}_{y \in \mathcal{Y}} f_y(\mathbf{x}) - \tau \log \pi_y \quad (15.141)$$

Alternatively, we can change the loss function used during training, by using the following **logit adjusted softmax cross-entropy loss**:

$$\ell(y, f(\mathbf{x})) = -\log \frac{e^{f_y(\mathbf{x}) + \tau \log \pi_y}}{\sum_{y'=1}^C e^{f_{y'}(\mathbf{x}) + \tau \log \pi_{y'}}} \quad (15.142)$$

This is like training with a predictor of the form $g_y(\mathbf{x}) = f_y(\mathbf{x}) + \tau \log \pi_y$, and then at test time using $\operatorname{argmax}_y f_y(\mathbf{x}) = \operatorname{argmax}_y g_y(\mathbf{x}) - \tau \log \pi_y$, as above.

We can also combine the above loss with a prior on the parameters and perform Bayesian inference, as we discuss below. (The use a non-standard likelihood can be justified using the generalized Bayesian inference framework, as discussed in Section 14.1.3.)

15.3.4 Parameter priors

As with linear regression, it is standard to use Gaussian priors for the weights in a logistic regression model. It is natural to set the prior mean to 0, to reflect the fact that the output could either

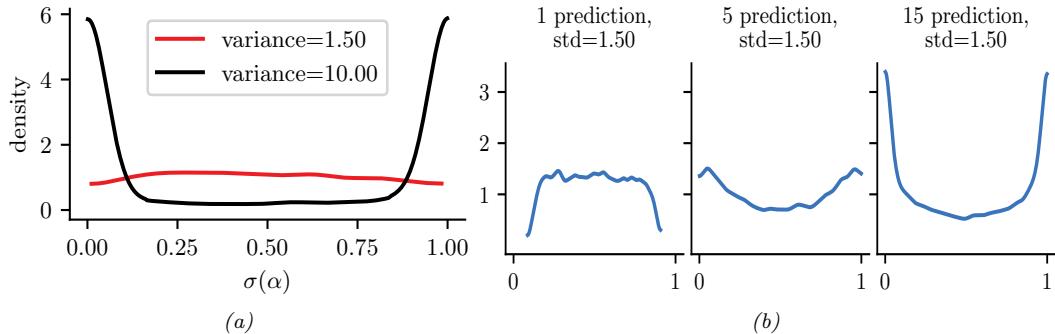


Figure 15.5: (a) Prior on logistic regression output when using $\mathcal{N}(0, \omega)$ prior for the offset term, for $\omega = 10$ or $\omega = 1.5$. Adapted from Figure 11.3 of [McE20]. Generated by [logreg_prior_offset.ipynb](#). (b) Distribution over the fraction of 1s we expect to see when using binary logistic regression applied to random binary feature vectors of increasing dimensionality. We use a $\mathcal{N}(0, 1.5)$ prior on the regression coefficients. Adapted from Figure 3 of [Gel+20]. Generated by [logreg_prior.ipynb](#).

increase or decrease in probability depending on the input. But how do we set the prior variance? It is tempting to use a large value, to approximate a uniform distribution, but this is a bad idea. To see why, consider a binary logistic regression model with just an offset term and no features:

$$p(y|\boldsymbol{\theta}) = \text{Ber}(y|\sigma(\alpha)) \quad (15.143)$$

$$p(\alpha) = \mathcal{N}(\alpha|0, \omega) \quad (15.144)$$

If we set the prior to the large value of $\omega = 10$, the implied prior for y is an extreme distribution, with most of its density near 0 or 1, as shown in Figure 15.5a. By contrast, if we use the smaller value of $\omega = 1.5$, we get a flatter distribution, as shown.

If we have input features, the problem gets a little trickier, since the magnitude of the logits will now depend on the number and distribution of the input variables. For example, suppose we generate N random binary vectors \mathbf{x}_n , each of dimension D , where $x_{nd} \sim \text{Ber}(p)$, where $p = 0.8$. We then compute $p(y_n = 1|\mathbf{x}_n) = \sigma(\boldsymbol{\beta}^\top \mathbf{x}_n)$, where $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{0}, 1.5\mathbf{I})$. We sample S values of $\boldsymbol{\beta}$, and for each one, we sample a vector of labels, $\mathbf{y}_{1:N,s}$ from the above distribution. We then compute the fraction of positive labels, $f_s = \frac{1}{N} \sum_{n=1}^N \mathbb{I}(y_{n,s} = 1)$. We plot the distribution of $\{f_s\}$ as a function of D in Figure 15.5b. We see that the induced prior is initially flat, but eventually becomes skewed towards the extreme values of 0 and 1. To avoid this, we should standardize the inputs, and scale the variance of the prior by $1/\sqrt{D}$. We can also use a heavier tailed distribution, such as a Cauchy or Student [Gel+08; GLM15], instead of the Gaussian prior.

15.3.5 Laplace approximation to the posterior

Unfortunately, we cannot compute the posterior analytically, unlike with linear regression, since there is no corresponding conjugate prior. (This mirrors the case with MLE, where we have a closed form solution for linear regression, but not for logistic regression.) Fortunately, there are a range of approximate inference methods we can use.

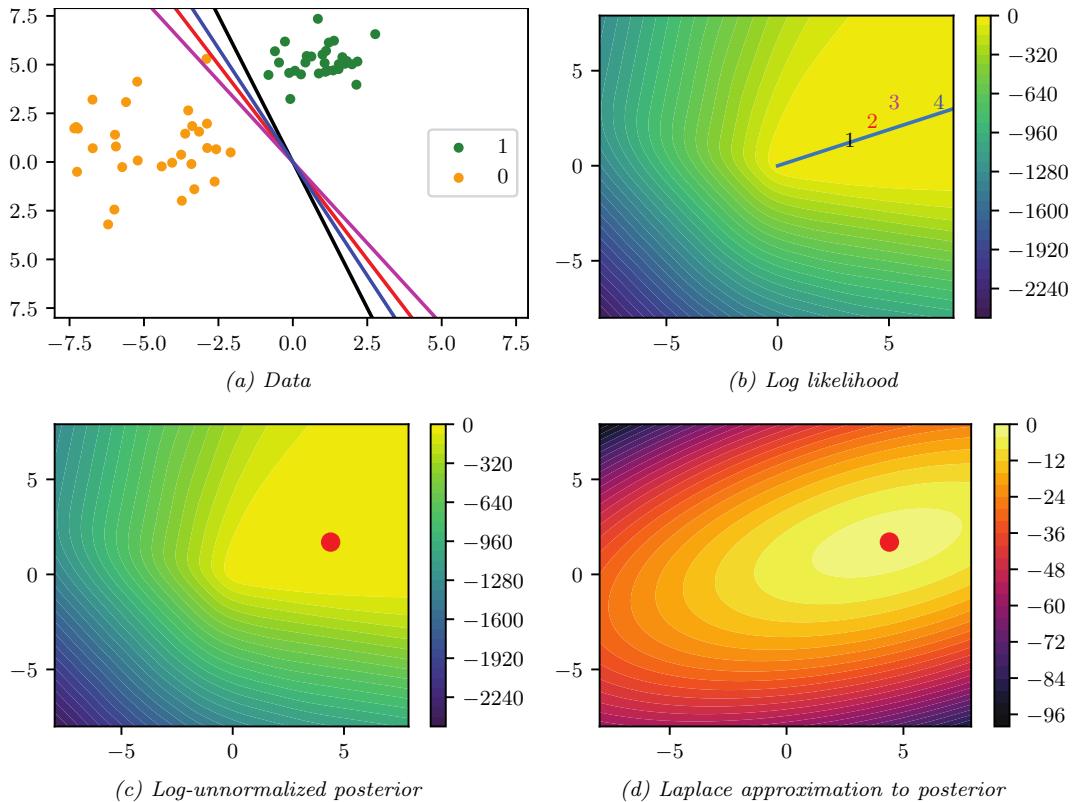


Figure 15.6: (a) Illustration of the data and some decision boundaries. (b) Log-likelihood for a logistic regression model. The line is drawn from the origin in the direction of the MLE (which is at infinity). The numbers correspond to 4 points in parameter space, corresponding to the colored lines in (a). (c) Unnormalized log posterior (assuming vague spherical prior). (d) Laplace approximation to posterior. Adapted from a figure by Mark Girolami. Generated by `logreg` `laplace` `demo.ipynb`.

In this section, we use the Laplace approximation. As we explain in Section 7.4.3, this approximates the posterior using a Gaussian. The mean of the Gaussian is equal to the MAP estimate $\hat{\mathbf{w}}$, and the covariance is equal to the inverse Hessian \mathbf{H} computed at the MAP estimate, i.e.,

$$p(\mathbf{w}|\mathcal{D}) \approx \mathcal{N}(\mathbf{w}|\hat{\mathbf{w}}, \mathbf{H}^{-1}), \hat{\mathbf{w}} = \arg \min -\log p(\mathbf{w}, \mathcal{D}), \mathbf{H} = -\nabla_{\mathbf{w}}^2 \log p(\mathbf{w}, \mathcal{D})|_{\hat{\mathbf{w}}} \quad (15.145)$$

We can find the mode using a standard optimization method, and we can then compute the Hessian at the mode analytically or using automatic differentiation.

As an example, consider the binary data illustrated in Figure 15.6(a). There are many parameter settings that correspond to lines that perfectly separate the training data; we show 4 example lines. For each decision boundary in Figure 15.6(a), we plot the corresponding parameter vector as point in the log likelihood surface in Figure 15.6(b). These parameters values are $\mathbf{w}_1 = (3, 1)$, $\mathbf{w}_2 = (4, 2)$, $\mathbf{w}_3 = (5, 3)$, and $\mathbf{w}_4 = (7, 3)$. These points all approximately satisfy $\mathbf{w}_i(1)/\mathbf{w}_i(2) \approx \hat{\mathbf{w}}_{\text{mle}}(1)/\hat{\mathbf{w}}_{\text{mle}}(2)$,

and hence are close to the orientation of the maximum likelihood decision boundary. The points are ordered by increasing weight norm (3.16, 4.47, 5.83, and 7.62). The unconstrained MLE has $\|\mathbf{w}\| = \infty$, so is infinitely far to the top right.

To ensure a unique solution, we use a (spherical) Gaussian prior centered at the origin, $\mathcal{N}(\mathbf{w}|\mathbf{0}, \sigma^2 \mathbf{I})$. The value of σ^2 controls the strength of the prior. If we set $\sigma^2 = \infty$, we force the MAP estimate to be $\mathbf{w} = \mathbf{0}$; this will result in maximally uncertain predictions, since all points \mathbf{x} will produce a predictive distribution of the form $p(y=1|\mathbf{x}) = 0.5$. If we set $\sigma^2 = 0$, the MAP estimate becomes the MLE, resulting in minimally uncertain predictions. (In particular, all positively labeled points will have $p(y=1|\mathbf{x}) = 1.0$, and all negatively labeled points will have $p(y=1|\mathbf{x}) = 0.0$, since the data is separable.) As a compromise (to make a nice illustration), we pick the value $\sigma^2 = 100$.

Multiplying this prior by the likelihood results in the unnormalized posterior shown in Figure 15.6(c). The MAP estimate is shown by the blue dot. The Laplace approximation to this posterior is shown in Figure 15.6(d). We see that it gets the mode correct (by construction), but the shape of the posterior is somewhat distorted. (The southwest-northeast orientation captures uncertainty about the magnitude of \mathbf{w} , and the southeast-northwest orientation captures uncertainty about the orientation of the decision boundary.)

15.3.6 Approximating the posterior predictive distribution

Next we need to convert the posterior over the parameters into a posterior over predictions, as follows:

$$p(y|\mathbf{x}, \mathcal{D}) = \int p(y|\mathbf{x}, \mathbf{w})p(\mathbf{w}|\mathcal{D})d\mathbf{w} \quad (15.146)$$

The simplest way to evaluate this integral is to use a Monte Carlo approximation. For example, in the case of binary logistic regression, we have

$$p(y=1|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \sigma(\mathbf{w}_s^\top \mathbf{x}) \quad (15.147)$$

where $\mathbf{w}_s \sim p(\mathbf{w}|\mathcal{D})$ are posterior samples.

However, we can also use deterministic approximations to the integral, which are often faster. Let $f_* = f(\mathbf{x}_*, \mathbf{w})$ be the predicted logits, before the sigmoid/softmax layer, given test point \mathbf{x}_* . If the posterior over the parameters is Gaussian, $p(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then the predictive distribution over logits is also Gaussian:

$$p(f_*|\mathbf{x}_*, \mathcal{D}) = \int \delta(f_* - f(\mathbf{x}_*, \mathbf{w}))\mathcal{N}(\mathbf{w}|\mathcal{D})d\mathbf{w} = \mathcal{N}(f_*|\boldsymbol{\mu}^\top \mathbf{x}_*, \mathbf{x}_*^\top \boldsymbol{\Sigma} \mathbf{x}_*) \triangleq \mathcal{N}(f_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \quad (15.148)$$

In the case of binary logistic regression, we can approximate the sigmoid with the probit function Φ (see Section 15.4), which allows us to solve the integral analytically:

$$p(y_*|\mathbf{x}_*) \approx \int \Phi(f_*)\mathcal{N}(f_*|\boldsymbol{\mu}_*, \sigma_*^2)df_* = \sigma\left(\frac{\boldsymbol{\mu}_*}{\sqrt{1 + \frac{\pi}{8}\sigma_*^2}}\right) \quad (15.149)$$

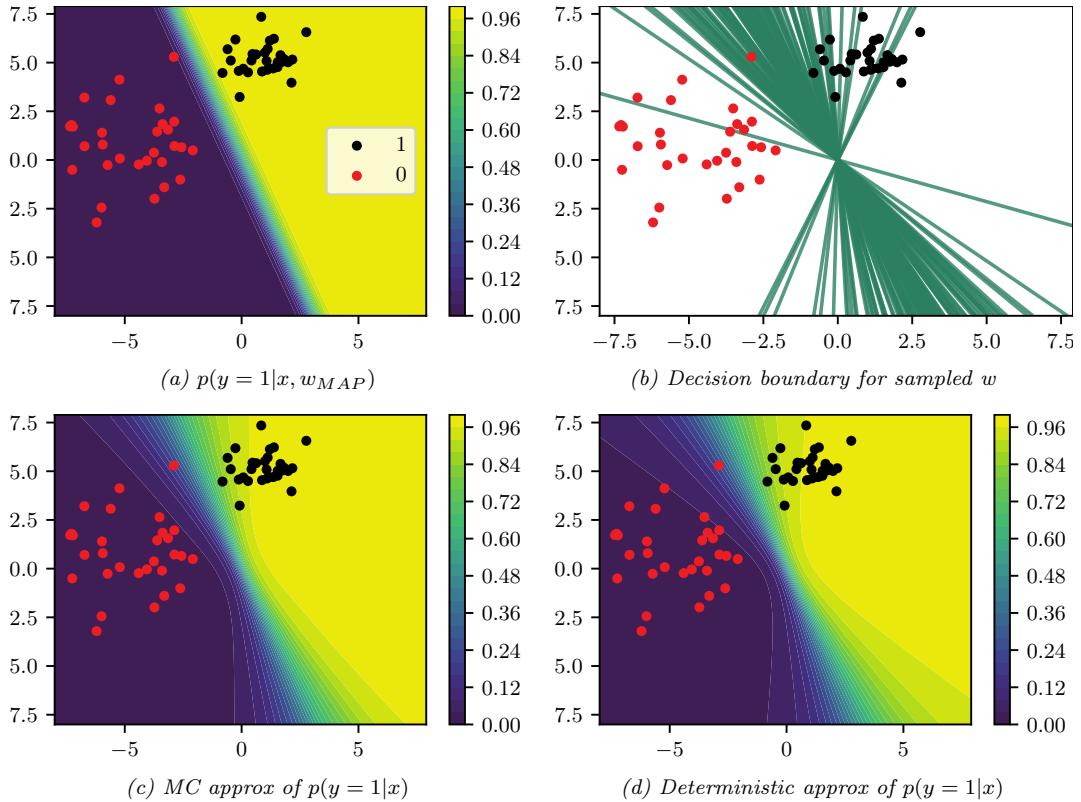


Figure 15.7: Posterior predictive distribution for a logistic regression model in 2d. (a) Contours of $p(y = 1|\mathbf{x}, \hat{\mathbf{w}}_{MAP})$. (b) Samples from the posterior predictive distribution. (c) Averaging over these samples. (d) Moderated output (probit approximation). Generated by [logreg_laplace_demo.ipynb](#).

This is called the **probit approximation** [SL90]. In [Gib97], a generalization to the multiclass case was provided. This is known as the **generalized probit approximation**, and has the form

$$p(\mathbf{y}_* | \mathbf{x}_*) \approx \int \text{softmax}(\mathbf{f}_*) \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) d\mathbf{f}_* = \text{softmax} \left(\left\{ \frac{\boldsymbol{\mu}_{*,c}}{\sqrt{1 + \frac{\pi}{8} \boldsymbol{\Sigma}_{*,cc}}} \right\} \right) \quad (15.150)$$

This ignores the correlations between the logits, because it only depends on the diagonal elements of $\boldsymbol{\Sigma}_*$. Nevertheless it can work well, even in the case of neural net classifiers [LIS20]. Another deterministic approximation, known as the **Laplace bridge**, is discussed in Section 17.3.10.2.

We now illustrate the posterior predictive for our binary example. Figure 15.7(a) shows the plugin approximation using the MAP estimate. We see that there is no uncertainty about the location of the decision boundary, even though we are generating probabilistic predictions over the labels. Figure 15.7(b) shows what happens when we plug in samples from the Gaussian posterior. Now we see that there is considerable uncertainty about the orientation of the “best” decision boundary. Figure 15.7(c) shows the average of these samples. By averaging over multiple predictions, we see

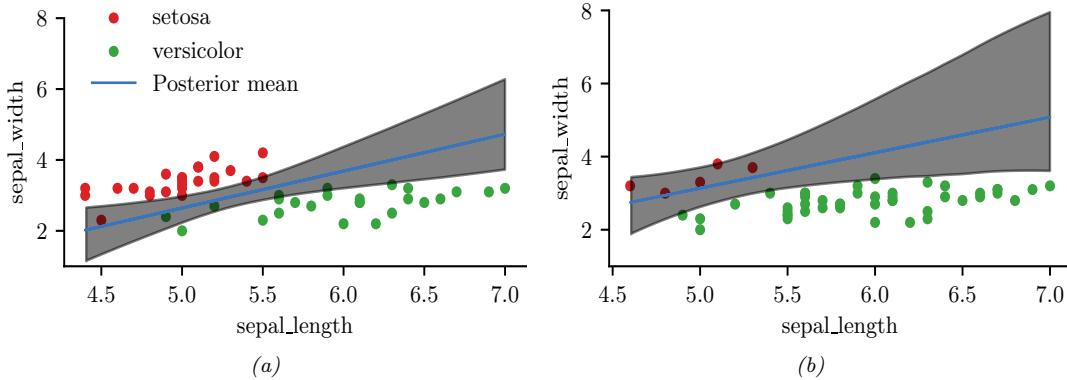


Figure 15.8: Illustration of the posterior over the decision boundary for classifying iris flowers (setosa vs versicolor) using 2 input features. (a) 25 examples per class. Adapted from Figure 4.5 of [Mar18]. (b) 5 examples of class 0, 45 examples of class 1. Adapted from Figure 4.8 of [Mar18]. Generated by [logreg_iris_bayes_2d.ipynb](#).

that the uncertainty in the decision boundary “splays out” as we move further from the training data. Figure 15.7(d) shows that the probit approximation gives very similar results to the Monte Carlo approximation.

15.3.7 MCMC inference

Markov chain Monte Carlo, or MCMC, is often considered the “gold standard” for approximate inference, since it makes no explicit assumptions about the form of the posterior. It is explained in depth in Chapter 12, but the output is a set of (correlated) samples from the posterior, which gives the following non-parametric approximation:

$$q(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s) \quad (15.151)$$

where $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$. Once we have the samples, we can plug them into Equation (15.147) to approximate the posterior predictive distribution.

A common MCMC method is known as Hamiltonian Monte Carlo (Section 12.5); this can leverage our ability to compute the gradient of the log joint, $\nabla_{\boldsymbol{\theta}} \log p(\mathcal{D}, \boldsymbol{\theta})$, for improved efficiency. Let us apply HMC to a 2-dimensional, 2-class version of the Iris classification problem, where we just use two input features, sepal length and sepal width, and two classes, Virginica and non-Virginica. The decision boundary is the set of points (x_1^*, x_2^*) such that $\sigma(b + w_1 x_1^* + w_2 x_2^*) = 0.5$. Such points must lie on the following line:

$$x_2^* = -\frac{b}{w_2} + \left(-\frac{w_1}{w_2} x_1^* \right) \quad (15.152)$$

We can therefore compute an MC approximation to the posterior over decision boundaries by sampling the parameters from the posterior, $(w_1, w_2, b) \sim p(\boldsymbol{\theta}|\mathcal{D})$, and plugging them into the above equation,

Dept. D_i	Gender G_i	# Admitted A_i	# Rejected R_i	# Applications N_i
A	male	512	313	825
A	female	89	19	108
B	male	353	207	560
B	female	17	8	25
C	male	120	205	325
C	female	202	391	593
D	male	138	279	417
D	female	131	244	375
E	male	53	138	191
E	female	94	299	393
F	male	22	351	373
F	female	24	317	341

Table 15.1: Admissions data for UC Berkeley from [BHO75].

to get $p(x_1^*, x_2^* | \mathcal{D})$. The results of this method (using a vague Gaussian prior for the parameters) are shown in Figure 15.8a. The solid line is the posterior mean, and the shaded interval is a 95% credible interval. As before, we see that the uncertainty about the location of the boundary is higher as we move away from the training data.

In Figure 15.8b, we show what happens to the decision boundary when we have unbalanced classes. We notice two things. First, the posterior uncertainty increases, because we have less data from the blue class. Second, we see that the posterior mean of the decision boundary shifts towards the class with less data. This follows from linear discriminant analysis, where one can show that changing the class prior changes the location of the decision boundary, so that more of the input space gets mapped to the class which is higher a priori. (See [Mur22, Sec 9.2] for details.)

15.3.8 Other approximate inference methods

There are many other approximate inference methods we can use, as we discuss in Part II. A common approach is variational inference (Section 10.1), which converts approximate inference into an optimization problem. It does this by choosing an approximate distribution $q(\mathbf{w}; \boldsymbol{\psi})$ and optimizing the variational parameters $\boldsymbol{\psi}$ to maximize the evidence lower bound (ELBO). This has the effect of making $q(\mathbf{w}; \boldsymbol{\psi}) \approx p(\mathbf{w} | \mathcal{D})$ in the sense that the KL divergence is small. There are several ways to tackle this: use a stochastic estimate of the ELBO (see Section 10.2.1), use the conditionally conjugate VI method of Supplementary Section 10.3.1.2, or use a “local” VI method that creates a quadratic lower bound to the logistic function (see Supplementary Section 15.1).

In the online setting, we can use assumed density filtering (ADF) to recursively compute a Gaussian approximate posterior $p(\mathbf{w} | \mathcal{D}_{1:t})$, as we discuss in Section 8.6.3.

15.3.9 Case study: is Berkeley admissions biased against women?

In this section, we consider a simple but interesting example of logistic regression from [McE20, Sec 11.1.4]. The question of interest is whether admission to graduate school at UC Berkeley is biased against women. The dataset comes from a famous paper [BHO75], which collected statistics for 6 departments for men and women. The data table only has 12 rows, shown in Table 15.1, although the total sample size (number of observations) is 4526. We conduct a regression analysis to try to determine if gender “causes” imbalanced admissions rates.

An obvious way to attempt to answer the question of interest is to fit a binomial logistic regression model, in which the outcome is the admissions rate, and the input is a binary variable representing the gender of each sample (male or female). One way to write this model is as follows:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.153)$$

$$\text{logit}(\mu_i) = \alpha + \beta \text{MALE}[i] \quad (15.154)$$

$$\alpha \sim \mathcal{N}(0, 10) \quad (15.155)$$

$$\beta \sim \mathcal{N}(0, 1.5) \quad (15.156)$$

Here A_i is the number of admissions for sample i , N_i is the number of applications, and $\text{MALE}[i] = 1$ iff the sample is male. So the log odds is α for female cases, and $\alpha + \beta$ for male candidates. (The choice of prior for these parameters is discussed in Section 15.3.4.)

The above formulation is asymmetric in the genders. In particular, the log odds for males has two random variables associated with it, and hence is a priori more uncertain. It is often better to rewrite the model in the following symmetric way:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.157)$$

$$\text{logit}(\mu_i) = \alpha_{\text{GENDER}[i]} \quad (15.158)$$

$$\alpha_j \sim \mathcal{N}(0, 1.5), j \in \{1, 2\} \quad (15.159)$$

Here $\text{GENDER}[i]$ is the gender (1 for male, 2 for female), so the log odds is α_1 for males and α_2 for females.

We can perform posterior inference using a variety of methods (see Chapter 7). Here we use HMC (Section 12.5). We find the 89% credible interval for α_1 is $[-0.29, 0.16]$ and for α_2 is $[-0.91, 0.75]$.² The corresponding distribution for the difference in probability, $\sigma(\alpha_1) - \sigma(\alpha_2)$, is $[0.12, 0.16]$, with a mean of 0.14. So it seems that Berkeley is biased in favor of men.

However, before jumping to conclusions, we should check if the model is any good. In Figure 15.9a, we plot the posterior predictive distribution, along with the original data. We see the model is a very bad fit to the data (the blue data dots are often outside the black predictive intervals). In particular, we see that the empirical admissions rate for women is actually higher in all the departments except for C and E, yet the model says that women should have a 14% lower chance of admission.

The trouble is that men and women did not apply to the same departments in equal amounts. Women tended not to apply to departments, like A and B, with high admissions rates, but instead applied more to departments, like F, with low admissions rates. So even though less women were accepted *overall*, within in each department, women tended to be accepted at about the same rate.

². McElreath uses 89% interval instead of 95% to emphasize the arbitrary nature of these values. The difference is insignificant.

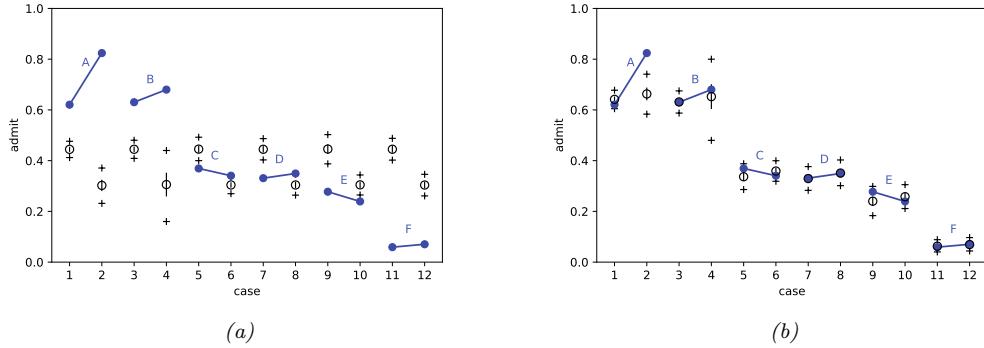


Figure 15.9: Blue dots are admission rates for each of the 6 departments (A-F) for males (left half of each dyad) and females (right half). The circle is the posterior mean of μ_i , the small vertical black lines indicate 1 standard deviation of μ_i . The + marks indicate 95% predictive interval for A_i . (a) Basic model, only taking gender into account. (b) Augmented model, adding department specific offsets. Adapted from Figure 11.5 of [McE20]. Generated by `logreg_ucb_admissions_numpyro.ipynb`.

We can get a better understanding if we consider the DAG in Figure 15.10a. This is intended to be a causal model of the relevant factors. We discuss causality in more detail in Chapter 36, but the basic idea should be clear from this picture. In particular, we see that there is an indirect causal path $G \rightarrow D \rightarrow A$ from gender to acceptance, so to infer the direct effect $G \rightarrow A$, we need to condition on D and close the indirect path. We can do this by adding department id as another feature:

$$A_i \sim \text{Bin}(N_i, \mu_i) \quad (15.160)$$

$$\text{logit}(\mu_i) = \alpha_{\text{GENDER}[i]} + \gamma_{\text{DEPT}[i]} \quad (15.161)$$

$$\alpha_j \sim \mathcal{N}(0, 1.5), j \in \{1, 2\} \quad (15.162)$$

$$\gamma_k \sim \mathcal{N}(0, 1.5), k \in \{1, \dots, 6\} \quad (15.163)$$

Here $j \in \{1, 2\}$ (for gender) and $k \in \{1, \dots, 6\}$ (for department). Note that there 12 parameters in this model, but each combination (slice of the data) has a fairly large sample size of data associated with it, as we see in Table 15.1.

In Figure 15.9b, we plot the posterior predictive distribution for this new model; we see the fit is now much better. We find the 89% credible interval for α_1 is $[-1.38, 0.35]$ and for α_2 is $[-1.31, 0.42]$. The corresponding distribution for the difference in probability, $\sigma(\alpha_1) - \sigma(\alpha_2)$, is $[-0.05, 0.01]$. So it seems that there is no bias after all.

However, the above conclusion is based on the correctness of the model in Figure 15.10a. What if there are **unobserved confounders** U , such as academic ability, influencing both admission rate and department choice? This hypothesis is shown in Figure 15.10b. In this case, conditioning on the collider D opens up a non-causal path between gender and admissions, $G \rightarrow D \leftarrow U \rightarrow A$. This invalidates any causal conclusions we may want to draw.

The point of this example is to serve as a cautionary tale to those trying to draw causal conclusions from predictive models. See Chapter 36 for more details.



Figure 15.10: Some possible causal models of admissions rates. G is gender, D is department, A is acceptance rate. (a) No hidden confounders. (b) Hidden confounder (small dot) affects both D and A . Generated by `logreg_ucb_admissions_numpyro.ipynb`.

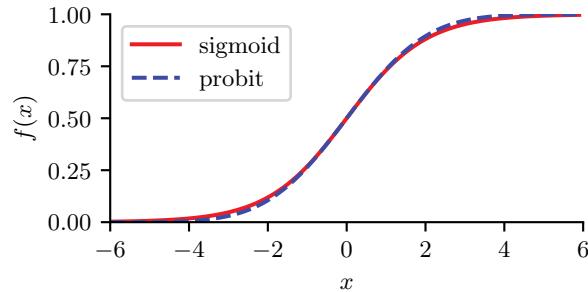


Figure 15.11: The logistic (sigmoid) function $\sigma(x)$ in solid red, with the Gaussian cdf function $\Phi(\lambda x)$ in dotted blue superimposed. Here $\lambda = \sqrt{\pi/8}$, which was chosen so that the derivatives of the two curves match at $x = 0$. Adapted from Figure 4.9 of [Bis06]. Generated by `probit_plot.ipynb`.

15.4 Probit regression

In this section, we discuss **probit regression**, which is similar to binary logistic regression except it uses $\mu_n = \Phi(a_n)$ instead of $\mu_n = \sigma(a_n)$ as the mean function, where Φ is the cdf of the standard normal, and $a_n = \mathbf{w}^\top \mathbf{x}_n$. The corresponding link function is therefore $a_n = \ell(\mu_n) = \Phi^{-1}(\mu_n)$; the inverse of the Gaussian cdf is known as the **probit function**.

The Gaussian cdf Φ is very similar to the logistic function, as shown in Figure 15.11. Thus probit regression and “regular” logistic regression behave very similarly. However, probit regression has some advantages. In particular, it has a simple interpretation as a latent variable model (see Section 15.4.1), which arises from the field of **choice theory** as studied in economics (see e.g., [Koo03]). This also simplifies the task of Bayesian parameter inference.

15.4.1 Latent variable interpretation

We can interpret $a_n = \mathbf{w}^\top \mathbf{x}_n$ as a factor that is proportional to how likely a person is respond positively (generate $y_n = 1$) given input \mathbf{x}_n . However, typically there are other unobserved factors that

influence someone's response. Let us model these hidden factors by Gaussian noise, $\epsilon_n \sim \mathcal{N}(0, 1)$. Let the combined preference for positive outcomes be represented by the latent variable $z_n = \mathbf{w}^\top \mathbf{x}_n + \epsilon_n$. We assume that the person will pick the positive label iff this latent factor is positive rather than negative, i.e.,

$$y_n = \mathbb{I}(z_n \geq 0) \quad (15.164)$$

When we marginalize out z_n , we recover the probit model:

$$p(y_n = 1 | \mathbf{x}_n, \mathbf{w}) = \int \mathbb{I}(z_n \geq 0) \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) dz_n \quad (15.165)$$

$$= p(\mathbf{w}^\top \mathbf{x}_n + \epsilon_n \geq 0) = p(\epsilon_n \geq -\mathbf{w}^\top \mathbf{x}_n) \quad (15.166)$$

$$= 1 - \Phi(-\mathbf{w}^\top \mathbf{x}_n) = \Phi(\mathbf{w}^\top \mathbf{x}_n) \quad (15.167)$$

Thus we can think of probit regression as a threshold function applied to noisy input.

We can interpret logistic regression in the same way. However, in that case the noise term ϵ_n comes from a **logistic distribution**, defined as follows:

$$f(y|\mu, s) \triangleq \frac{e^{-\frac{y-\mu}{s}}}{s(1+e^{-\frac{y-\mu}{s}})^2} = \frac{1}{4s} \operatorname{sech}^2\left(\frac{y-\mu}{s}\right) \quad (15.168)$$

where the mean is μ and the variance is $\frac{s^2\pi^2}{3}$. The cdf of this distribution is given by

$$F(y|\mu, s) = \frac{1}{1+e^{-\frac{y-\mu}{s}}} \quad (15.169)$$

It is clear that if we use logistic noise with $\mu = 0$ and $s = 1$ we recover logistic regression. However, it is computationally easier to deal with Gaussian noise, as we show below.

15.4.2 Maximum likelihood estimation

In this section, we discuss some methods for fitting probit regression using MLE.

15.4.2.1 MLE using SGD

We can find the MLE for probit regression using standard gradient methods. Let $\mu_n = \mathbf{w}^\top \mathbf{x}_n$, and let $\tilde{y}_n \in \{-1, +1\}$. Then the gradient of the log-likelihood for a single example n is given by

$$\mathbf{g}_n \triangleq \frac{d}{d\mathbf{w}} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \frac{d\mu_n}{d\mathbf{w}} \frac{d}{d\mu_n} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = \mathbf{x}_n \frac{\tilde{y}_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \quad (15.170)$$

where ϕ is the standard normal pdf, and Φ is its cdf. Similarly, the Hessian for a single case is given by

$$\mathbf{H}_n = \frac{d}{d\mathbf{w}^2} \log p(\tilde{y}_n | \mathbf{w}^\top \mathbf{x}_n) = -\mathbf{x}_n \left(\frac{\phi(\mu_n)^2}{\Phi(\tilde{y}_n \mu_n)^2} + \frac{\tilde{y}_n \mu_n \phi(\mu_n)}{\Phi(\tilde{y}_n \mu_n)} \right) \mathbf{x}_n^\top \quad (15.171)$$

This can be passed to any gradient-based optimizer.

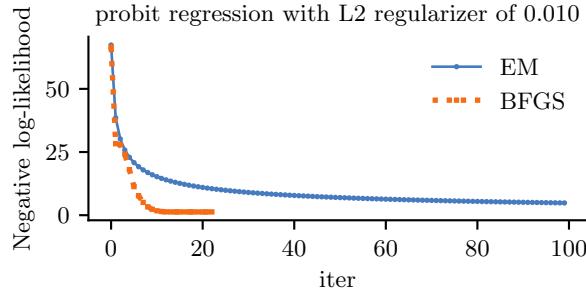


Figure 15.12: Fitting a probit regression model in 2d using a quasi-Newton method or EM. Generated by `probit_reg_demo.ipynb`.

15.4.2.2 MLE using EM

We can use the latent variable interpretation of probit regression to derive an elegant EM algorithm for fitting the model. The complete data log likelihood has the following form, assuming a $\mathcal{N}(\mathbf{0}, \mathbf{V}_0)$ prior on \mathbf{w} :

$$\ell(\mathbf{z}, \mathbf{w} | \mathbf{V}_0) = \log p(\mathbf{y} | \mathbf{z}) + \log \mathcal{N}(\mathbf{z} | \mathbf{X}\mathbf{w}, \mathbf{I}) + \log \mathcal{N}(\mathbf{w} | \mathbf{0}, \mathbf{V}_0) \quad (15.172)$$

$$= \sum_n \log p(y_n | z_n) - \frac{1}{2} (\mathbf{z} - \mathbf{X}\mathbf{w})^\top (\mathbf{z} - \mathbf{X}\mathbf{w}) - \frac{1}{2} \mathbf{w}^\top \mathbf{V}_0^{-1} \mathbf{w} \quad (15.173)$$

The posterior in the E step is a **truncated Gaussian**:

$$p(z_n | y_n, \mathbf{x}_n, \mathbf{w}) = \begin{cases} \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n > 0) & \text{if } y_n = 1 \\ \mathcal{N}(z_n | \mathbf{w}^\top \mathbf{x}_n, 1) \mathbb{I}(z_n < 0) & \text{if } y_n = 0 \end{cases} \quad (15.174)$$

In Equation (15.173), we see that \mathbf{w} only depends linearly on \mathbf{z} , so we just need to compute $\mathbb{E}[z_n | y_n, \mathbf{x}_n, \mathbf{w}]$, so we just need to compute the posterior mean. One can show that this is given by

$$\mathbb{E}[z_n | \mathbf{w}, \mathbf{x}_n] = \begin{cases} \mu_n + \frac{\phi(\mu_n)}{1 - \Phi(-\mu_n)} = \mu_n + \frac{\phi(\mu_n)}{\Phi(\mu_n)} & \text{if } y_n = 1 \\ \mu_n - \frac{\phi(\mu_n)}{\Phi(-\mu_n)} = \mu_n - \frac{\phi(\mu_n)}{1 - \Phi(\mu_n)} & \text{if } y_n = 0 \end{cases} \quad (15.175)$$

where $\mu_n = \mathbf{w}^\top \mathbf{x}_n$.

In the M step, we estimate \mathbf{w} using ridge regression, where $\boldsymbol{\mu} = \mathbb{E}[\mathbf{z}]$ is the output we are trying to predict. Specifically, we have

$$\hat{\mathbf{w}} = (\mathbf{V}_0^{-1} + \mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{\mu} \quad (15.176)$$

The EM algorithm is simple, but can be much slower than direct gradient methods, as illustrated in Figure 15.12. This is because the posterior entropy in the E step is quite high, since we only observe that z is positive or negative, but are given no information from the likelihood about its magnitude. Using a stronger regularizer can help speed convergence, because it constrains the range of plausible z values. In addition, one can use various speedup tricks, such as data augmentation [DM01].

15.4.3 Bayesian inference

It is possible to use the latent variable formulation of probit regression in Section 15.4.2.2 to derive a simple Gibbs sampling algorithm for approximating the posterior $p(\mathbf{w}|\mathcal{D})$ (see e.g., [AC93; HH06]).

The key idea is to use an auxiliary latent variable, which, when conditioned on, makes the whole model a conjugate linear-Gaussian model. The full conditional for the latent variables is given by

$$p(z_i|y_i, \mathbf{x}_i, \mathbf{w}) = \begin{cases} \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i > 0) & \text{if } y_i = 1 \\ \mathcal{N}(z_i|\mathbf{w}^T \mathbf{x}_i, 1)\mathbb{I}(z_i < 0) & \text{if } y_i = 0 \end{cases} \quad (15.177)$$

Thus the posterior is a truncated Gaussian. We can sample from a truncated Gaussian, $\mathcal{N}(z|\mu, \sigma)\mathbb{I}(a \leq z \leq b)$ in two steps: first sample $u \sim U(\Phi((a - \mu)/\sigma), \Phi((b - \mu)/\sigma))$, then set $z = \mu + \sigma\Phi^{-1}(u)$ [Rob95a].

The full conditional for the parameters is given by

$$p(\mathbf{w}|\mathcal{D}, \mathbf{z}, \boldsymbol{\lambda}) = \mathcal{N}(\mathbf{w}_N, \mathbf{V}_N) \quad (15.178)$$

$$\mathbf{V}_N = (\mathbf{V}_0^{-1} + \mathbf{X}^T \mathbf{X})^{-1} \quad (15.179)$$

$$\mathbf{w}_N = \mathbf{V}_N(\mathbf{V}_0^{-1}\mathbf{w}_0 + \mathbf{X}^T \mathbf{z}) \quad (15.180)$$

For further details, see e.g., [AC93; FSF10]. It is also possible to use variational Bayes, which tends to be much faster (see e.g., [GR06a; FDZ19]).

15.4.4 Ordinal probit regression

One advantage of the latent variable interpretation of probit regression is that it is easy to extend to the case where the response variable is ordered in some way, such as the outputs low, medium, and high. This is called **ordinal regression**. The basic idea is as follows. If there are C output values, we introduce $C + 1$ thresholds γ_j and set

$$y_n = j \quad \text{if} \quad \gamma_{j-1} < z_n \leq \gamma_j \quad (15.181)$$

where $\gamma_0 \leq \dots \leq \gamma_C$. For identifiability reasons, we set $\gamma_0 = -\infty$, $\gamma_1 = 0$ and $\gamma_C = \infty$. For example, if $C = 2$, this reduces to the standard binary probit model, whereby $z_n < 0$ produces $y_n = 0$ and $z_n \geq 0$ produces $y_n = 1$. If $C = 3$, we partition the real line into 3 intervals: $(-\infty, 0]$, $(0, \gamma_2]$, (γ_2, ∞) . We can vary the parameter γ_2 to ensure the right relative amount of probability mass falls in each interval, so as to match the empirical frequencies of each class label. See e.g., [AC93] for further details.

Finding the MLEs for this model is a bit trickier than for binary probit regression, since we need to optimize for \mathbf{w} and $\boldsymbol{\gamma}$, and the latter must obey an ordering constraint. See e.g., [KL09] for an approach based on EM. It is also possible to derive a simple Gibbs sampling algorithm for this model (see e.g., [Hof09, p216]).

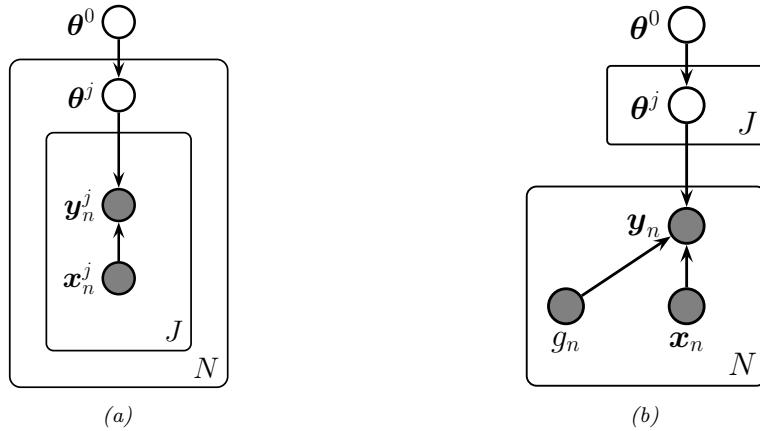


Figure 15.13: Hierarchical Bayesian discriminative models with J groups. (a) Nested formulation. (b) Non-nested formulation, with group indicator $g_n \in \{1, \dots, J\}$.

15.4.5 Multinomial probit models

Now consider the case where the response variable can take on C unordered categorical values, $y_n \in \{1, \dots, C\}$. The **multinomial probit** model is defined as follows:

$$z_{nc} = \mathbf{w}_c^\top \mathbf{x}_{nc} + \epsilon_{nc} \quad (15.182)$$

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \quad (15.183)$$

$$y_n = \arg \max_c z_{nc} \quad (15.184)$$

See e.g., [DE04; GR06b; Sco09; FSF10] for more details on the model and its connection to multinomial logistic regression.

If instead of setting $y_n = \arg \max_c z_{ic}$ we use $y_{nc} = \mathbb{I}(z_{nc} > 0)$, we get a model known as **multivariate probit**, which is one way to model C correlated binary outcomes (see e.g., [TMD12]).

15.5 Multilevel (hierarchical) GLMs

Suppose we have a set of J related datasets, each of which contains a series of N_j datapoints $\mathcal{D}_j = \{(\mathbf{x}_n^j, \mathbf{y}_n^j) : n = 1 : N_j\}$. There are 3 main ways to fit models in such a setting: we could fit J separate models, $p(\mathbf{y}|\mathbf{x}; \mathcal{D}_j)$, which might result in overfitting if some \mathcal{D}_j are small; we could pool all the data to get $\mathcal{D} = \cup_{j=1}^J \mathcal{D}_j$ and fit a single model, $p(\mathbf{y}|\mathbf{x}; \mathcal{D})$, which might result in underfitting; or we can use a **hierarchical Bayesian model**, also called a **multilevel model** or **partially pooled model**, in which we assume each group has its own parameters, θ^j , but that these have something in common, as modeled by a shared global prior $p(\theta^0)$. (Note that each group could be a single individual.) The overall model has the form

$$p(\theta^{0:J}, \mathcal{D}) = p(\theta^0) \prod_{j=1}^J \left[p(\theta^j | \theta^0) \prod_{n=1}^{N_j} p(\mathbf{y}_n^j | \mathbf{x}_n^j, \theta^j) \right] \quad (15.185)$$

See Figure 15.13a, which represents the model using nested plate notation.

It is often more convenient to represent the model as in Figure 15.13b, which eliminates the nested plates (and hence the double indexing of variables) by associating a group indicator variable $g_n \in \{1, \dots, J\}$, which specifies which set of parameters to use for each datapoint. Thus the model now has the form

$$p(\boldsymbol{\theta}^{0:J}, \mathcal{D}) = p(\boldsymbol{\theta}^0) \left[\prod_{j=1}^J p(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0) \right] \left[\prod_{n=1}^N p(\mathbf{y}_n | \mathbf{x}_n, g_n, \boldsymbol{\theta}) \right] \quad (15.186)$$

where

$$p(\mathbf{y}_n | \mathbf{x}_n, g_n, \boldsymbol{\theta}) = \prod_{j=1}^J p(\mathbf{y}_n | \mathbf{x}_n, \boldsymbol{\theta}^j)^{\mathbb{I}(g_n=j)} \quad (15.187)$$

If the likelihood function is a GLM, this hierarchical model is called a **hierarchical GLM** [LN96]. This class of models is very widely used in applied statistics. For much more details, see e.g., [GH07; GHV20b; Gel+22].

15.5.1 Generalized linear mixed models (GLMMs)

Suppose that the prior on the per-group parameters is Gaussian, so $p(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0) = \mathcal{N}(\boldsymbol{\theta}^j | \boldsymbol{\theta}^0, \boldsymbol{\Sigma}^j)$. If we have a GLM likelihood, the model becomes

$$p(\mathbf{y}_n | \mathbf{x}_n, g_n = j, \boldsymbol{\theta}) = p(\mathbf{y}_n | \ell(\eta_n)) \quad (15.188)$$

$$\eta_n = \mathbf{x}_n^\top \boldsymbol{\theta}^j = \mathbf{x}_n^\top (\boldsymbol{\theta}^0 + \boldsymbol{\epsilon}^j) = \mathbf{x}_n^\top \boldsymbol{\theta}^0 + \mathbf{x}_n^\top \boldsymbol{\epsilon}^j \quad (15.189)$$

where ℓ is the link function, and $\boldsymbol{\epsilon}^j \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. This is known as a **generalized linear mixed model (GLMM)** or **mixed effects model**. The shared (common) parameters $\boldsymbol{\theta}^0$ are called **fixed effects**, and the group-specific offsets $\boldsymbol{\epsilon}^j$ are called **random effects**.³ We can see that the random effects model group-specific deviations or idiosyncrasies away from the shared fixed parameters. Furthermore, we see that the random effects are correlated, which allows us to model dependencies between the observations that would not be captured by a standard GLM.

For model fitting, we can use any of the Bayesian inference methods that we discussed in Section 15.1.4.

15.5.2 Example: radon regression

In this section, we give an example of a hierarchical Bayesian linear regression model. We apply it to a simplified version of the **radon** example from [Gel+14a, Sec 9.4].

Radon is known to be the highest cause of lung cancer in non-smokers, so reducing it where possible is desirable. To help with this, we fit a regression model, that predicts the (log) radon level as a function of the location of the house, as represented by a categorical feature indicating its county, and

3. Note that there are multiple definitions of the terms “fixed effects” and random effects”, as explained in this blog post by Andrew Gelman: https://statmodeling.stat.columbia.edu/2005/01/25/why_i_dont_use/.

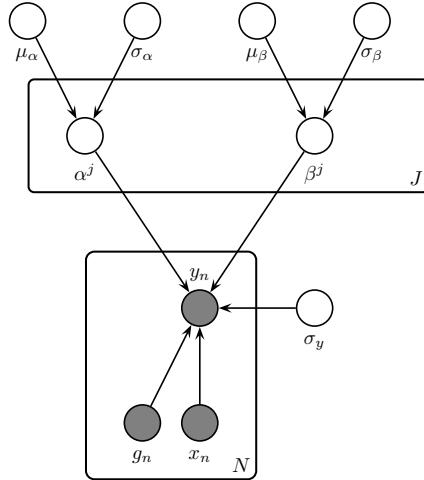


Figure 15.14: A hierarchical Bayesian linear regression model for the radon problem.

a binary feature representing whether the house has a basement or not. We use a dataset consisting of $J = 85$ counties in Minnesota; each county has between 2 and 80 measurements.

We assume the following likelihood:

$$p(y_n|x_n, g_n = j, \boldsymbol{\theta}) = \mathcal{N}(y_n|\alpha_j + \beta_j x_n, \sigma_y^2) \quad (15.190)$$

where $g_n \in \{1, \dots, J\}$ is the county for house i , and $x_n \in \{0, 1\}$ indicates if the floor is at level 0 (i.e., in the basement) or level 1 (i.e., above ground). Intuitively we expect the radon levels to be lower in houses without basements, since they are more insulated from the earth which is the source of the radon.

Since some counties have very few datapoints, we use a hierarchical prior in which we assume $\alpha_j \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2)$, and $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$. We use weak priors for the parameters: $\mu_\alpha \sim \mathcal{N}(0, 1)$, $\mu_\beta \sim \mathcal{N}(0, 1)$, $\sigma_\alpha \sim \mathcal{C}_+(1)$, $\sigma_\beta \sim \mathcal{C}_+(1)$, $\sigma_y \sim \mathcal{C}_+(1)$. See Figure 15.14 for the graphical model.

15.5.2.1 Posterior inference

Figure 15.15 shows the posterior marginals for μ_α , μ_β , α_j and β_j . We see that μ_β is close to -0.6 with high probability, which confirms our suspicion that having $x = 1$ (i.e., no basement) decreases the amount of radon in the house. We also see that the distribution of the α_j parameters is quite variable, due to different base rates across the counties.

Figure 15.16 shows predictions from the hierarchical and non-hierarchical model for 3 different counties. We see that the predictions from the hierarchical model are more consistent across counties, and work well even if there are no examples of certain feature combinations for a given county (e.g., there are no houses without basements in the sample from Cass county). If we sample data from the posterior predictive distribution, and compare it to the real data, we find that the RMSE is 0.13 for the non-hierarchical model and 0.08 for the hierarchical model, indicating that the latter fits better.

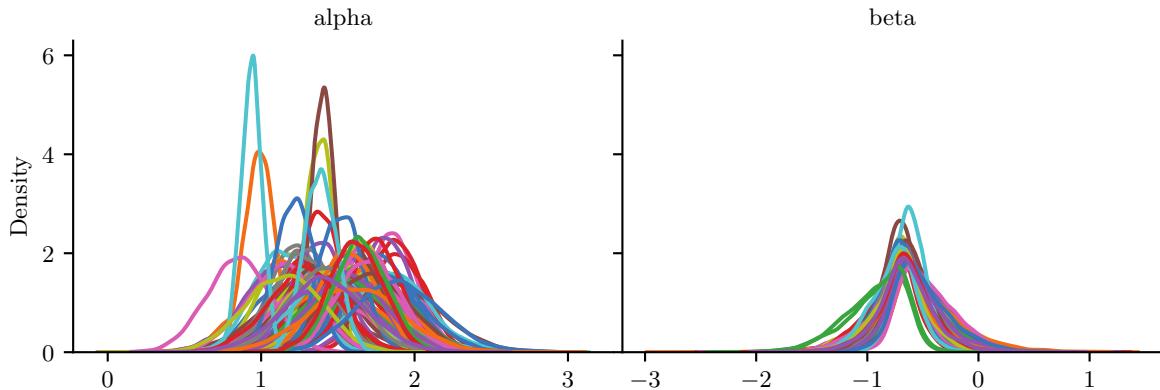


Figure 15.15: Posterior marginals for α_j and β_j for each county j in the radon model. Generated by `linreg_hierarchical_non_centered.ipynb`.

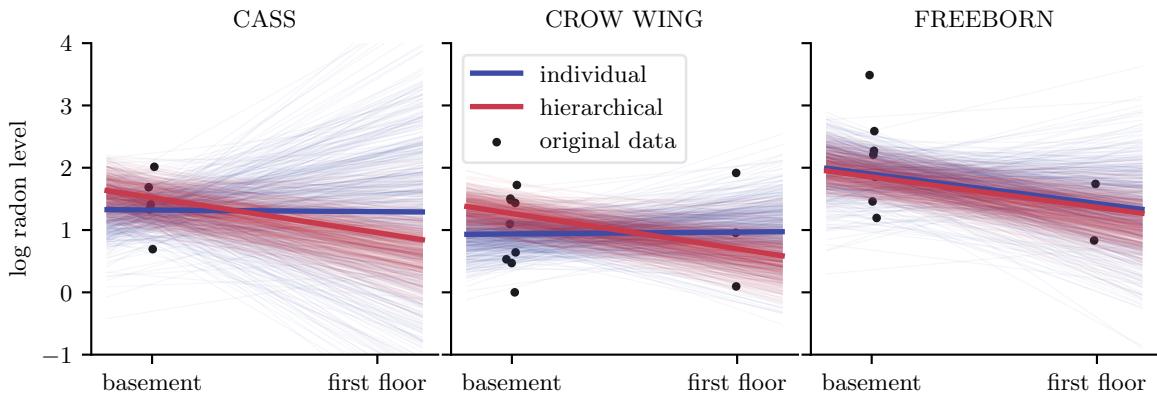


Figure 15.16: Predictions from the radon model for 3 different counties in Minnesota. Black dots are observed datapoints. Red represents results of hierarchical (shared) prior, blue represents results of non-hierarchical prior. Thick lines are the result of using the posterior mean, thin lines are the result of using posterior samples. Generated by `linreg_hierarchical_non_centered.ipynb`.

15.5.2.2 Non-centered parameterization

One problem that frequently arises in hierarchical models is that the parameters be very correlated. This can cause computational problems when performing inference.

Figure 15.17a gives an example where we plot $p(\beta_j, \sigma_\beta | \mathcal{D})$ for some specific county j . If we believe that σ_β is large, then β_c is “allowed” to vary a lot, and we get the broad distribution at the top of the figure. However, if we believe that σ_β is small, then β_j is constrained to be close to the global prior mean of μ_β , so we get the narrow distribution at the bottom of the figure. This is often called **Neal’s funnel**, after a paper by Radford Neal [Nea03]. It is difficult for many algorithms (especially sampling algorithms) to explore parts of parameter space at the bottom of the funnel. This is evident from the marginal posterior for σ_β shown (as a histogram) on the right hand side of the plot: we see

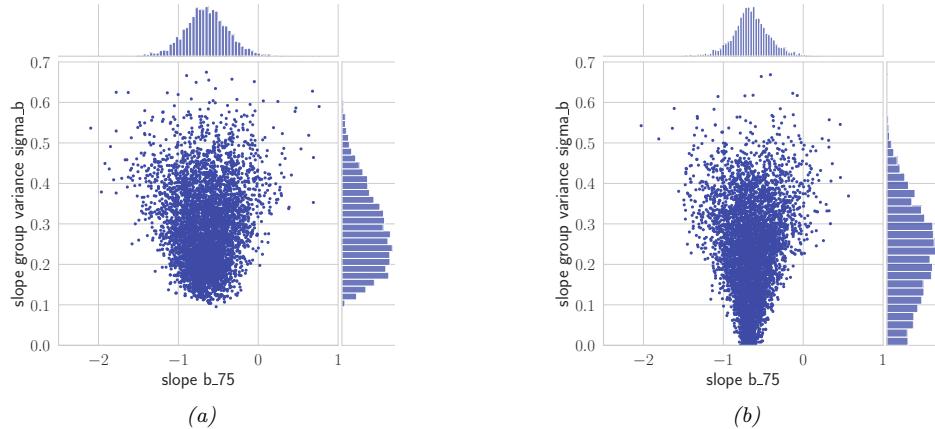


Figure 15.17: (a) Bivariate posterior $p(\beta_j, \sigma_\beta | \mathcal{D})$ for the hierarchical radon model for county $j = 75$ using centered parameterization. (b) Similar to (a) except we plot $p(\tilde{\beta}_j, \sigma_\beta | \mathcal{D})$ for the non-centered parameterization. Generated by [linreg_hierarchical_non_centered.ipynb](#).

that it excludes the interval $[0, 0.1]$, thus ruling out models in which we shrink β_j all the way to 0. In cases where a covariate has no useful predictive role, we would like to be able to induce sparsity, so we need to overcome this problem.

A simple solution to this is to use a **non-centered parameterization** [PR03]. That is, we replace $\beta_j \sim \mathcal{N}(\mu_\beta, \sigma_\beta^2)$ with $\beta_j = \mu_\beta + \tilde{\beta}_j \sigma_\beta$, where $\tilde{\beta}_j \sim \mathcal{N}(0, 1)$ represents the *offset* from the global mean, μ_β . The correlation between $\tilde{\beta}_j$ and σ_β is much less, as shown in Figure 15.17b. See Section 12.6.5 for more details.

16 Deep neural networks

16.1 Introduction

The term “**deep neural network**” or **DNN**, in its modern usage, refers to any kind of differentiable function that can be expressed as a **computation graph**, where the nodes are primitive operations (like matrix multiplication), and edges represent numeric data in the form of vectors, matrices, or tensors. In its simplest form, this graph can be constructed as a linear series of nodes or “**layers**”. The term “deep” refers to models with many such layers.

In Section 16.2 we discuss some of the basic building blocks (node types) that are used in the field. In Section 16.3 we give examples of common architectures which are constructed from these building blocks. In Section 6.2 we show how we can efficiently compute the gradient of functions defined on such graphs. If the function computes the scalar loss of the model’s predictions given a training set, we can pass this gradient to an optimization routine, such as those discussed in Chapter 6, in order to fit the model. Fitting such models to data is called “**deep learning**”.

We can combine DNNs with probabilistic models in two different ways. The first is to use them to define nonlinear functions which are used inside conditional distributions. For example, we may construct a classifier using $p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\mathbf{x}; \boldsymbol{\theta})))$, where $f(\mathbf{x}; \boldsymbol{\theta})$ is a neural network that maps inputs \mathbf{x} and parameters $\boldsymbol{\theta}$ to output logits. Or we may construct a joint probability distribution over multiple variables using a directed graphical model (Chapter 4) where each CPD $p(\mathbf{x}_i|\text{pa}(\mathbf{x}_i))$ is a DNN. This lets us construct expressive probability models.

The other way we can combine DNNs and probabilistic models is to use DNNs to approximate the posterior distribution, i.e., we learn a function f to compute $q(\mathbf{z}|f(\mathcal{D}; \boldsymbol{\phi}))$, where \mathbf{z} are the hidden variables (latents and/or parameters), \mathcal{D} are the observed variables (data), f is an **inference network**, and $\boldsymbol{\phi}$ are its parameters; for details, see Section 10.1.5. Note that in this latter, setting the joint model $p(\mathbf{z}, \mathcal{D})$ may be a “traditional” model without any “neural” components. For example, it could be a complex simulator. Thus the DNN is just used for computational purposes, not statistical/modeling purposes.

More details on DNNs can be found in such books as [Zha+20a; Cho21; Gér19; GBC16; Raf22], as well as a multitude of online courses. For a more theoretical treatment, see e.g., [Ber+21; Cal20; Aro+21; RY21].

16.2 Building blocks of differentiable circuits

In this section we discuss some common building blocks used in constructing neural networks. We denote the input to a block as \mathbf{x} and the output as \mathbf{y} .



Figure 16.1: An artificial “neuron”, the most basic building block of a DNN. (a) The output y is a weighted combination of the inputs \mathbf{x} , where the weights vector is denoted by \mathbf{w} . (b) Alternative depiction of the neuron’s behavior. The bias term b can be emulated by defining $w_N = b$ and $X_N = 1$.

16.2.1 Linear layers

The most basic building block of a DNN is a single “**neuron**”, which corresponds to a real-valued signal y computed by multiplying a vector-valued input signal \mathbf{x} by a weight vector \mathbf{w} , and then adding a bias term b . That is,

$$y = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^\top \mathbf{x} + b \quad (16.1)$$

where $\theta = (\mathbf{w}, b)$ are the parameters for the function f . This is depicted in Figure 16.1. (The bias term is omitted for clarity.)

It is common to group a set of neurons together into a **layer**. We can then represent the activations of a layer with D units as a vector $\mathbf{z} \in \mathbb{R}^D$. We can transform an input vector of activations \mathbf{x} into an output vector \mathbf{y} by multiplying by a weight matrix \mathbf{W} , an adding an offset vector or bias term \mathbf{b} to get

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (16.2)$$

where $\theta = (\mathbf{W}, \mathbf{b})$ are the parameters for the function f . This is called a **linear layer**, or **fully connected layer**.

It is common to prepend the bias vector onto the first column of the weight matrix, and to append a 1 to the vector \mathbf{x} , so that we can write this more compactly as $\mathbf{x} = \tilde{\mathbf{W}}^\top \tilde{\mathbf{x}}$, where $\tilde{\mathbf{W}} = [\mathbf{W}, \mathbf{b}]$ and $\tilde{\mathbf{x}} = [\mathbf{x}, 1]$. This allows us to ignore the bias term from our notation if we want to.

16.2.2 Nonlinearities

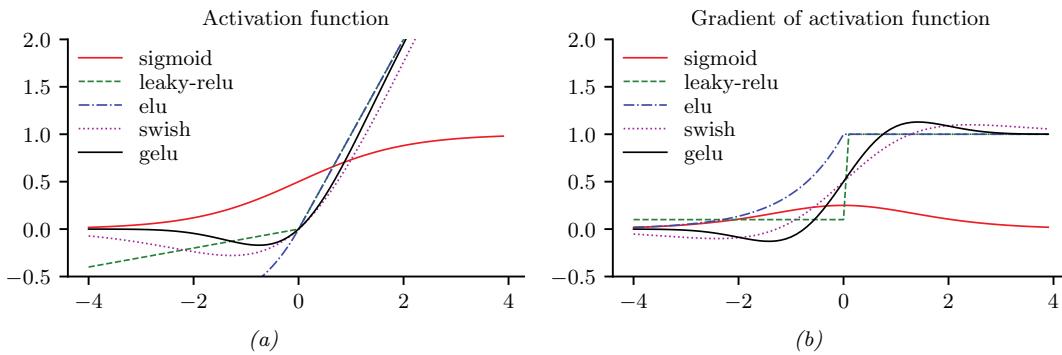
A stack of linear layers is equivalent to a single linear layer where we multiply together all the weight matrices. To get more expressive power we can transform each layer by passing it elementwise (pointwise) through a nonlinear function called an **activation function**. This is denoted by

$$\mathbf{y} = \varphi(\mathbf{x}) = [\varphi(x_1), \dots, \varphi(x_D)] \quad (16.3)$$

See Table 16.1 for a list of some common activation functions, and Figure 16.2 for a visualization. For more details, see e.g., [Mur22, Sec 13.2.3].

Name	Definition	Range	Reference
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$[0, 1]$	
Hyperbolic tangent	$\tanh(a) = 2\sigma(2a) - 1$	$[-1, 1]$	
Softplus	$\sigma_+(a) = \log(1 + e^a)$	$[0, \infty]$	[GBB11]
Rectified linear unit	$\text{ReLU}(a) = \max(a, 0)$	$[0, \infty]$	[GBB11; KSH12a]
Leaky ReLU	$\max(a, 0) + \alpha \min(a, 0)$	$[-\infty, \infty]$	[MHN13]
Exponential linear unit	$\max(a, 0) + \min(\alpha(e^a - 1), 0)$	$[-\infty, \infty]$	[CUH16]
Swish	$a\sigma(a)$	$[-\infty, \infty]$	[RZL17]
GELU	$a\Phi(a)$	$[-\infty, \infty]$	[HG16]
Sine	$\sin(a)$	$[-1, 1]$	[Sit+20]

Table 16.1: List of some popular activation functions for neural networks.

Figure 16.2: (a) Some popular activation functions. “ReLU” stands for “restricted linear unit”. “GELU” stands for “Gaussian error linear unit”. (b) Plot of their gradients. Generated by [activation_fun_deriv.ipynb](#).

16.2.3 Convolutional layers

When dealing with image data, we can apply the same weight matrix to each local patch of the image, in order to reduce the number of parameters. If we “slide” this weight matrix over the image and add up the results, we get a technique known as **convolution**; in this case the weight matrix is often called a “**kernel**” or “**filter**”.

More precisely, let $\mathbf{X} \in \mathbb{R}^{H \times W}$ be the input image, and $\mathbf{W} \in \mathbb{R}^{h \times w}$ be the kernel. The output is denoted by $\mathbf{Z} = \mathbf{X} * \mathbf{W}$, where (ignoring boundary conditions) we have the following:¹

$$Z_{i,j} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} x_{i+u, j+v} w_{u,v} \quad (16.4)$$

Essentially we compare a local patch of \mathbf{x} , of size $h \times w$ and centered at (i, j) , to the filter \mathbf{w} ; the output just measures how similar the input patch is to the filter. We can define convolution in 1d or

1. Note that, technically speaking, we are using **cross correlation** rather than convolution. However, these terms are used interchangeably in deep learning.

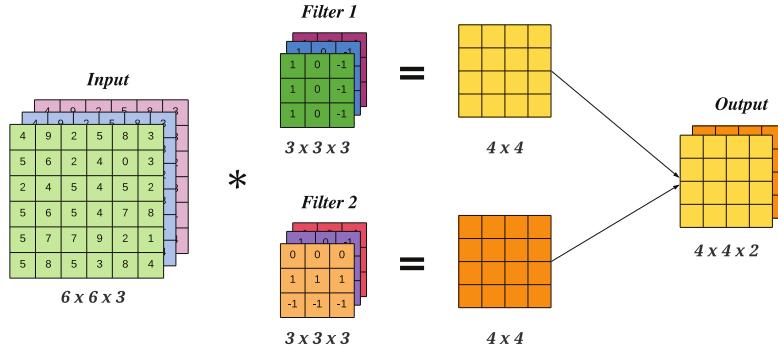


Figure 16.3: A 2d convolutional layer with 3 input channels and 2 output channels. The kernel has size 3×3 and we use stride 1 with 0 padding, so the 6×6 input gets mapped to the 4×4 output.

3d in an analogous manner. Note that the spatial size of the outputs may be smaller than inputs, due to boundary effects, although this can be solved by using **padding**. See [Mur22, Sec 14.2.1] for more details.

We can repeat this process for multiple layers of inputs, and by using multiple filters, we can generate multiple layers of output. In general, if we have C input channels, and we want to map it to D output (feature) channels, then we define D kernels, each of size $h \times w \times C$, where h, w are the height and width of the kernel. The d 'th output feature map is obtained by convolving all C input feature maps with the d 'th kernel, and then adding up the results elementwise:

$$z_{i,j,d} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} \sum_{c=0}^{C-1} x_{i+u,j+v,c} w_{u,v,c,d} \quad (16.5)$$

This is called a **convolutional layer**, and is illustrated in Figure 16.3.

The advantage of a convolutional layer compared to using a linear layer is that the weights of the kernel are shared across locations in the input. Thus if a pattern in the input shifts locations, the corresponding output activation will also shift. This is called **shift equivariance**. In some cases, we want the output to be the same, no matter where the input pattern occurs; this is called **shift invariance**, and can be obtained by using a **pooling layer**, which computes the maximum or average value in each local patch of the input. (Note that pooling layers have no free (learnable) parameters.) Other forms of invariance can also be captured by neural networks (see e.g., [CW16; FWW21]).

16.2.4 Residual (skip) connections

If we stack a large number of nonlinear layers together, the signal may get squashed to zero or may blow up to infinity, depending on the magnitude of the weights, and the nature of the nonlinearities. Similar problems can plague gradients that are passed backwards through the network (see Section 6.2). To reduce the effect of this we can add **skip connections**, also called **residual connections**, which

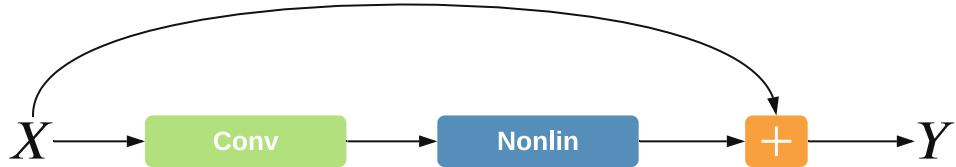


Figure 16.4: A residual connection around a convolutional layer.

allow the signal to skip one or more layers, which prevents it from being modified. For example, Figure 16.4 illustrates a network that computes

$$\mathbf{y} = f(\mathbf{x}; \mathbf{W}) = \varphi(\text{conv}(\mathbf{x}; \mathbf{W})) + \mathbf{x} \quad (16.6)$$

Now the convolutional layer only needs to learn an offset or residual to add (or subtract) to the input to match the desired output, rather than predicting the output directly. Such residuals are often small in size, and hence are easier to learn using neurons with weights that are bounded (e.g., close to 1).

16.2.5 Normalization layers

To learn an input-output mapping, it is often best if the inputs are standardized, meaning that they have zero mean and unit standard deviation. This ensures that the required magnitude of the weights is small, and comparable across dimensions. To ensure that the internal activations have this property, it is common to add **normalization layers**.

The most common approach is to use **batch normalization (BN)** [IS15]. However this relies on having access to a batch of $B > 1$ input examples. Various alternatives have been proposed to overcome the need of having an input batch, such as **layer normalization** [BKH16], **instance normalization** [UVL16], **group normalization** [WH18], **filter response normalization** [SK20], etc. More details can be found in [Mur22, Sec 14.2.4].

16.2.6 Dropout layers

Neural networks often have millions of parameters, and thus can sometimes overfit, especially when trained on small datasets. There are many ways to ameliorate this effect, such as applying regularizers to the weights, or adopting a fully Bayesian approach (see Chapter 17). Another common heuristic is known as **dropout** [Sri+14a], in which edges are randomly omitted each time the network is used, as illustrated in Figure 16.5. More precisely, if w_{lij} is the weight of the edge from node i in layer $l - 1$ to node j in layer $l + 1$, then we replace it with $\theta_{lij} = w_{lij}\epsilon_{li}$, where $\epsilon_{li} \sim \text{Ber}(1 - p)$, where p is the drop probability, and $1 - p$ is the keep probability. Thus if we sample $\epsilon_{li} = 0$, then all of the weights going out of unit i in layer $l - 1$ into any j in layer l will be set to 0.

During training, the gradients will be zero for the weights connected to a neuron which has been switched “off”. However, since we resample ϵ_{lij} every time the network is used, different combinations

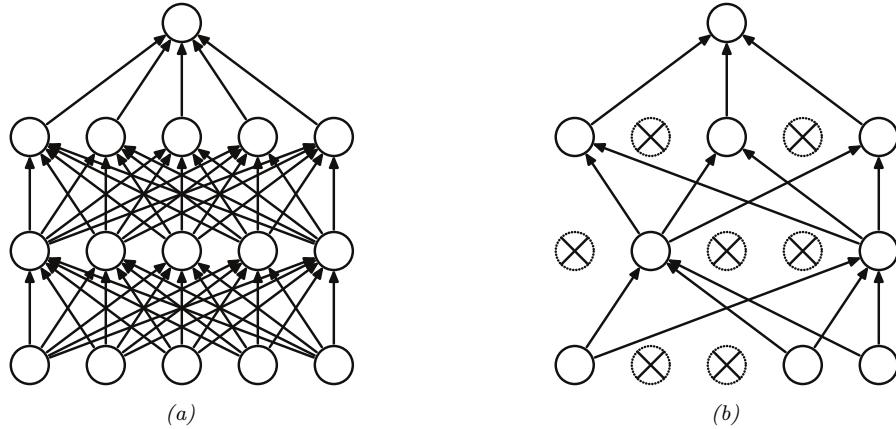


Figure 16.5: Illustration of dropout. (a) A standard neural net with 2 hidden layers. (b) An example of a thinned net produced by applying dropout with $p = 0.5$. Units that have been dropped out are marked with an x . From Figure 1 of [Sri+14a]. Used with kind permission of Geoff Hinton.

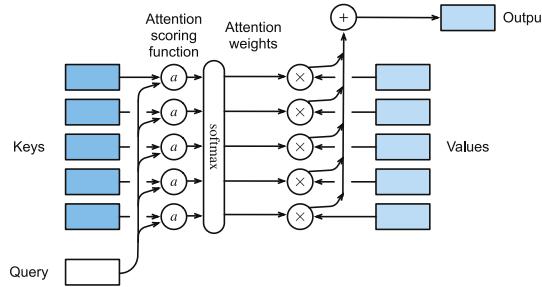


Figure 16.6: Attention layer. (a) Mapping a single query \mathbf{q} to a single output, given a set of keys and values. From Figure 10.3.1 of [Zha+20a]. Used with kind permission of Aston Zhang.

of weights will be updated on each step. The result is an **ensemble** of networks, each with slightly different sparse graph structures.

At test time, we usually turn the dropout noise off, so the model acts deterministically. To ensure the weights have the same expectation at test time as they did during training (so the input activation to the neurons is the same, on average), at test time we should use $\mathbb{E}[\theta_{lij}] = w_{lij}\mathbb{E}[\epsilon_{li}]$. For Bernoulli noise, we have $\mathbb{E}[\epsilon] = 1 - p$, so we should multiply the weights by the keep probability, $1 - p$, before making predictions. We can, however, use dropout at test time if we wish. This is called **Monte Carlo dropout** (see Section 17.3.1).

16.2.7 Attention layers

In all of the neural networks we have considered so far, the hidden activations are a linear combination of the input activations, followed by a nonlinearity: $\mathbf{Z} = \varphi(\mathbf{XW})$, where $\mathbf{X} \in \mathbb{R}^{n \times d}$ are the hidden

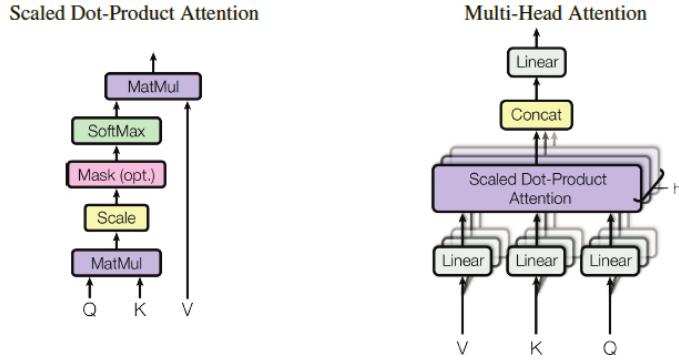


Figure 16.7: (a) Scaled dot-product attention in matrix form. (b) Multi-head attention. From Figure 2 of [Vas+17b]. Used with kind permission of Ashish Vaswani.

feature vectors, and $\mathbf{W} \in \mathbb{R}^{d \times d_v}$ are a fixed set of weights that are learned on a training set to produce $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$ outputs. However, we can imagine a more flexible model in which the weights depend on the inputs, i.e., $\mathbf{Z} = \varphi(\mathbf{X}\mathbf{W}(\mathbf{X}))$, where $\mathbf{W}(\mathbf{X})$ is a function to be defined below. This kind of **multiplicative interaction** is called **attention**.

We can better understand attention by comparing it to non-parametric kernel based prediction methods, such as Gaussian processes (Chapter 18). In this approach we compare the input query $\mathbf{x} \in \mathbb{R}^d$ to each of the training examples $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ using a kernel to get a vector of similarity scores, $\boldsymbol{\alpha} = [\mathcal{K}(\mathbf{x}, \mathbf{x}_i)]_{i=1}^n$. We then use this to retrieve a weighted combination of the corresponding m target values $\mathbf{y}_i \in \mathbb{R}^{d_v}$ to compute the predicted output, as follows:

$$\hat{\mathbf{y}} = \sum_{i=1}^n \alpha_i \mathbf{y}_i \quad (16.7)$$

See Section 18.3.7 for details.

We can make a differentiable and parametric version of this as follows (see [Tsa+19] for details). First we replace the stored examples matrix \mathbf{X} with a learned embedding, to create a set of stored **keys**, $\mathbf{K} = \mathbf{X}\mathbf{W}_k \in \mathbb{R}^{n \times d_k}$. Similarly we replace the stored output matrix \mathbf{Y} with a learned embedding, to create a set of stored **values**, $\mathbf{V} = \mathbf{Y}\mathbf{W}_v \in \mathbb{R}^{n \times d_v}$. Finally we embed the input to create a **query**, $\mathbf{q} = \mathbf{W}_q \mathbf{x} \in \mathbb{R}^{d_k}$. The parameters to be learned are the three embedding matrices.

Next, we replace fixed kernel function with a soft **attention layer**. More precisely, we define the weighted output for query \mathbf{q} to be

$$\text{Attn}(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_n, \mathbf{v}_n)) = \text{Attn}(\mathbf{q}, (\mathbf{k}_{1:n}, \mathbf{v}_{1:n})) = \sum_{i=1}^n \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) \mathbf{v}_i \quad (16.8)$$

where $\alpha_i(\mathbf{q}, \mathbf{k}_{1:n})$ is the i 'th **attention weight**; these weights satisfy $0 \leq \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) \leq 1$ for each i and $\sum_i \alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) = 1$.

The attention weights can be computed from an **attention score** function $a(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$, that computes the similarity of query \mathbf{q} to key \mathbf{k}_i . For example, we can use (scaled) **dot product**

attention, which has the form

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d_k} \quad (16.9)$$

(The scaling by $\sqrt{d_k}$ is to reduce the dependence of the output on the dimensionality of the vectors.) Given the scores, we can compute the attention weights using the softmax function:

$$\alpha_i(\mathbf{q}, \mathbf{k}_{1:n}) = \text{softmax}_i([a(\mathbf{q}, \mathbf{k}_1), \dots, a(\mathbf{q}, \mathbf{k}_n)]) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^n \exp(a(\mathbf{q}, \mathbf{k}_j))} \quad (16.10)$$

See Figure 16.6 for an illustration.

In some cases, we want to restrict attention to a subset of the dictionary, corresponding to valid entries. For example, we might want to pad sequences to a fixed length (for efficient minibatching), in which case we should “mask out” the padded locations. This is called **masked attention**. We can implement this efficiently by setting the attention score for the masked entries to a large negative number, such as -10^6 , so that the corresponding softmax weights will be 0.

For efficiently, we usually compute all n vectors in parallel. Let the corresponding matrices of queries, keys and values be denoted by $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{n \times d_k}$, $\mathbf{V} \in \mathbb{R}^{n \times d_v}$. Let

$$\mathbf{z}_j = \sum_{i=1}^n \alpha_i(\mathbf{q}_j, \mathbf{K}) \mathbf{v}_i \quad (16.11)$$

be the j 'th output corresponding to the j 'th query. We can compute all outputs $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$ in parallel using

$$\mathbf{Z} = \text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \quad (16.12)$$

where the softmax function softmax is applied row-wise. See Figure 16.7 (left) for an illustration.

To increase the flexibility of the model, we often use a **multi-head attention** layer, as illustrated in Figure 16.7 (right). Let the i 'th head be

$$\mathbf{h}_i = \text{Attn}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \quad (16.13)$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$ are linear projection matrices. We define the output of the MHA layer to be

$$\mathbf{Z} = \text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\mathbf{h}_1, \dots, \mathbf{h}_h) \mathbf{W}^O \quad (16.14)$$

where h is the number of heads, and $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$. Having multiple heads can increase performance of the layer, in the event that some of the weight matrices are poorly initialized; after training, we can often remove all but one of the heads [MLN19].

When the output of one attention layer is used as input to another, the method is called **self-attention**. This is the basis of the transformer model, which we discuss in Section 16.3.5.

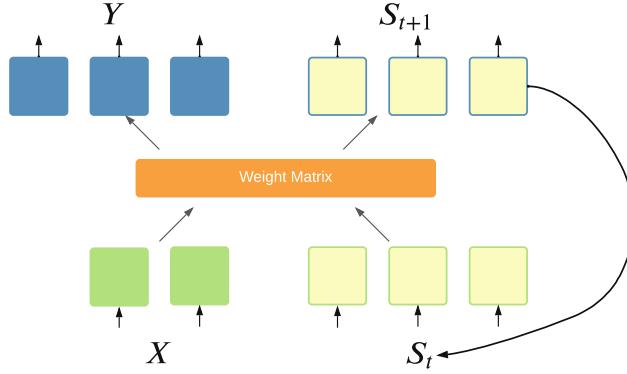


Figure 16.8: Recurrent layer.

16.2.8 Recurrent layers

We can make the model be **stateful** by augmenting the input \mathbf{x} with the current state \mathbf{s}_t , and then computing the output and the new state using some kind of function:

$$(\mathbf{y}, \mathbf{s}_{t+1}) = f(\mathbf{x}, \mathbf{s}_t) \quad (16.15)$$

This is called a **recurrent layer**, as shown in Figure 16.8. This forms the basis of **recurrent neural networks**, discussed in Section 16.3.4. In a vanilla RNN, the function f is a simple MLP, but it may also use attention (Section 16.2.7).

16.2.9 Multiplicative layers

In this section, we discuss **multiplicative layers**, which are useful for combining different information sources. Our presentation follows [Jay+20].

Suppose we have inputs $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$. In a linear layer (and, by extension, convolutional layers), it is common to concatenate the inputs to get $f(\mathbf{x}, \mathbf{z}) = \mathbf{W}[\mathbf{x}; \mathbf{z}] + \mathbf{b}$, where $\mathbf{W} \in \mathbb{R}^{k \times (m+n)}$ and $\mathbf{b} \in \mathbb{R}^k$. We can increase the expressive power of the model by using **multiplicative interactions**, such as the following **bilinear form**:

$$f(\mathbf{x}, \mathbf{z}) = \mathbf{z}^\top \mathbb{W} \mathbf{x} + \mathbf{U} \mathbf{z} + \mathbf{V} \mathbf{x} + \mathbf{b} \quad (16.16)$$

where $\mathbb{W} \in \mathbb{R}^{m \times n \times k}$ is a weight tensor, defined such that

$$(\mathbf{z}^\top \mathbb{W} \mathbf{x})_k = \sum_{ij} \mathbf{z}_i \mathbb{W}_{ijk} \mathbf{x}_j \quad (16.17)$$

That is, the k 'th entry of the output is the weighted inner product of \mathbf{z} and \mathbf{x} , where the weight matrix is the k 'th “slice” of \mathbb{W} . The other parameters have size $\mathbf{U} \in \mathbb{R}^{k \times m}$, $\mathbf{V} \in \mathbb{R}^{k \times n}$, and $\mathbf{b} \in \mathbb{R}^k$.

This formulation includes many interesting special cases. In particular, a **hypernetwork** [HDL17] can be viewed in this way. A hypernetwork is a neural network that generates parameters for another

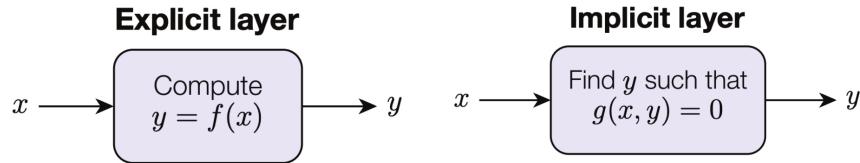


Figure 16.9: Explicit vs implicit layers.

neural network. In particular, we replace $f(\mathbf{x}; \boldsymbol{\theta})$ with $f(\mathbf{x}; g(\mathbf{z}; \boldsymbol{\phi}))$. If f and g are affine, this is equivalent to a multiplicative layer. To see this, let $\mathbf{W}' = \mathbf{z}^\top \mathbf{W} + \mathbf{V}$ and $\mathbf{b}' = \mathbf{U}\mathbf{z} + \mathbf{b}$. If we define $g(\mathbf{z}; \boldsymbol{\Phi}) = [\mathbf{W}', \mathbf{b}']$, and $f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}'\mathbf{x} + \mathbf{b}'$, we recover Equation (16.16).

We can also view the gating layers used in RNNs (Section 16.3.4) as a form of multiplicative interaction. In particular, if the hypernetwork computes the diagonal matrix $\mathbf{W}' = \sigma(\mathbf{z}^\top \mathbf{W} + \mathbf{V}) = \text{diag}(a_1, \dots, a_n)$, then we can define $f(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta}) = \mathbf{a}(\mathbf{z}) \odot \mathbf{x}$, which is the standard gating mechanism. Attention mechanisms (Section 16.2.7) are also a form of multiplicative interaction, although they involve three-way interactions, between query, key, and value.

Another variant arises if the hypernetwork just computes a scalar weight for each channel of a convolutional layer, plus a bias term:

$$f(\mathbf{x}, \mathbf{z}) = \mathbf{a}(\mathbf{z}) \odot \mathbf{x} + \mathbf{b}(\mathbf{z}) \quad (16.18)$$

This is called **FiLM**, which stands for “feature-wise linear modulation” [Per+18]. For a detailed tutorial on the FiLM layer and its many applications, see <https://distill.pub/2018/feature-wise-transformations>.

16.2.10 Implicit layers

So far we have focused on **explicit layers**, which specify how to transform the input to the output using $\mathbf{y} = f(\mathbf{x})$. We can also define **implicit layers**, which specify the output indirectly, in terms of a constraint function:

$$\mathbf{y} \in \underset{\mathbf{y}}{\operatorname{argmin}} f(\mathbf{x}, \mathbf{y}) \text{ such that } g(\mathbf{x}, \mathbf{y}) = 0 \quad (16.19)$$

The details on how to find a solution to this constrained optimization problem can vary depending on the problem. For example, we may need to run an inner optimization routine, or call a differential equation solver. The main advantage of this approach is that the inner computations do not need to be stored explicitly, which saves a lot of memory. Furthermore, once the solution has been found, we can propagate gradients through the whole layer, by leveraging the implicit function theorem. This lets us use higher level primitives inside an end-to-end framework. For more details, see [GHC21] and <http://implicit-layers-tutorial.org/>.

16.3 Canonical examples of neural networks

In this section, we give several “canonical” examples of neural network architectures that are widely used for different tasks.

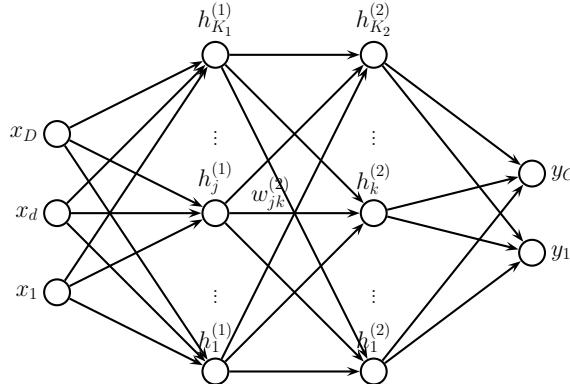


Figure 16.10: A feedforward neural network with D inputs, K_1 hidden units in layer 1, K_2 hidden units in layer 2, and C outputs. $w_{jk}^{(l)}$ is the weight of the connection from node j in layer $l-1$ to node k in layer l .

16.3.1 Multilayer perceptrons (MLPs)

A **multilayer perceptron (MLP)**, also called a **feedforward neural network (FFNN)**, is one of the simplest kinds of neural networks. It consists of a series of L linear layers, combined with elementwise nonlinearities:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_L \varphi_L (\mathbf{W}_{L-1} \varphi_{L-1} (\cdots \varphi_1 (\mathbf{W}_1 \mathbf{x}) \cdots)) \quad (16.20)$$

For example, Figure 16.10 shows an MLP with 1 input layer of D units, 2 hidden layers of K_1 and K_2 units, and 1 output layer with C units. The k 'th hidden unit in layer l is given by

$$h_k^{(l)} = \varphi_l \left(b_k^{(l)} + \sum_{j=1}^{K_{l-1}} w_{jk}^{(l)} h_j^{(l-1)} \right) \quad (16.21)$$

where φ_l is the nonlinear activation function at layer l .

For a classification problem, the final nonlinearity is usually the softmax function. However, it is also common for the final layer to have linear activations, in which case the outputs are interpreted as logits; the loss function used during training then converts to (log) probabilities internally.

We can also use MLPs for regression. Figure 16.11 shows how we can make a model for **heteroskedastic** nonlinear regression. (The term “heteroskedastic” just means that the predicted output variance is input-dependent, rather than a constant.) This function has two outputs which compute $f_\mu(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}, \boldsymbol{\theta}]$ and $f_\sigma(\mathbf{x}) = \sqrt{\mathbb{V}[y|\mathbf{x}, \boldsymbol{\theta}]}$. We can share most of the layers (and hence parameters) between these two functions by using a common “**backbone**” and two output “**heads**”, as shown in Figure 16.11. For the μ head, we use a linear activation, $\varphi(a) = a$. For the σ head, we use a softplus activation, $\varphi(a) = \sigma_+(a) = \log(1 + e^a)$. If we use linear heads and a nonlinear backbone, the overall model is given by

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y | \mathbf{w}_\mu^\top f(\mathbf{x}; \mathbf{w}_{\text{shared}}), \sigma_+(\mathbf{w}_\sigma^\top f(\mathbf{x}; \mathbf{w}_{\text{shared}}))) \quad (16.22)$$

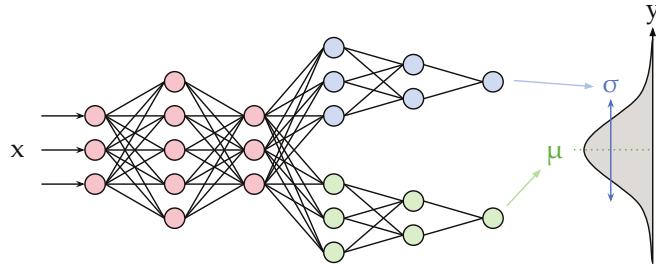


Figure 16.11: Illustration of an MLP with a shared “backbone” and two output “heads”, one for predicting the mean and one for predicting the variance. From <https://brendanhasz.github.io/2019/07/23/bayesian-density-net.html>. Used with kind permission of Brendan Hasz.

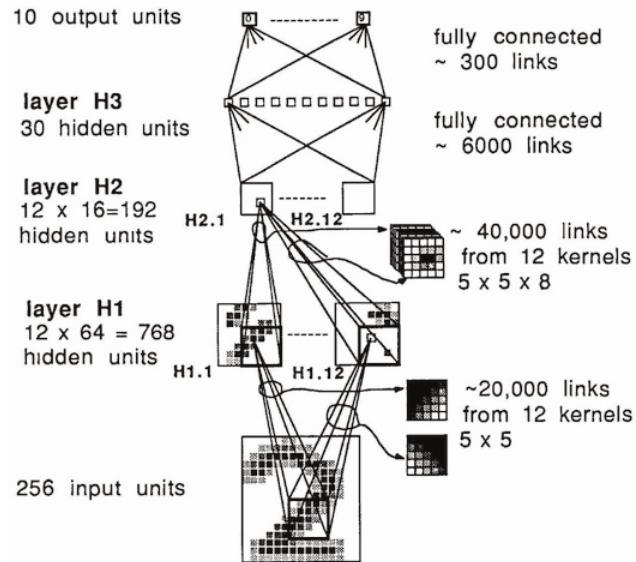


Figure 16.12: One of the first CNNs ever created, for classifying MNIST images. From Figure 3 of [LeC+89]. For a “modern” implementation, see [lecun1989.ipynb](#).

16.3.2 Convolutional neural networks (CNNs)

A vanilla **convolutional neural network** or **CNN** consists of a series of convolutional layers, pooling layers, linear layers, and nonlinearities. See Figure 16.12 for an example. More sophisticated architectures, such as the **ResNet** model [He+16a; He+16b], add skip (residual) connections, normalization layers, etc. The **ConvNeXt** model of [Liu+22b] is considered the current (as of February 2022) state of the art CNN architecture for a wide variety of vision tasks. See e.g., [Mur22, Ch.14] for more details on CNNs.

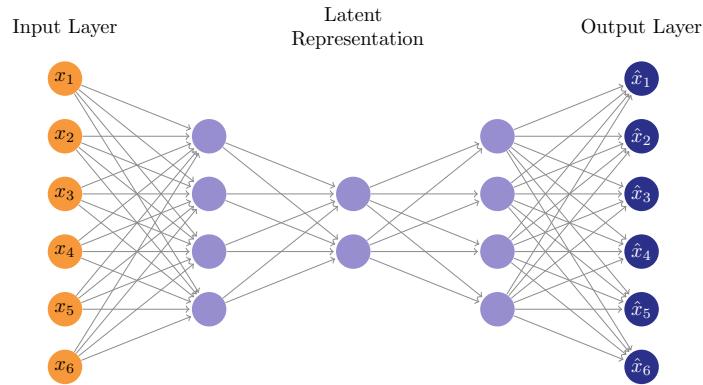


Figure 16.13: Illustration of an autoencoder with 3 hidden layers.

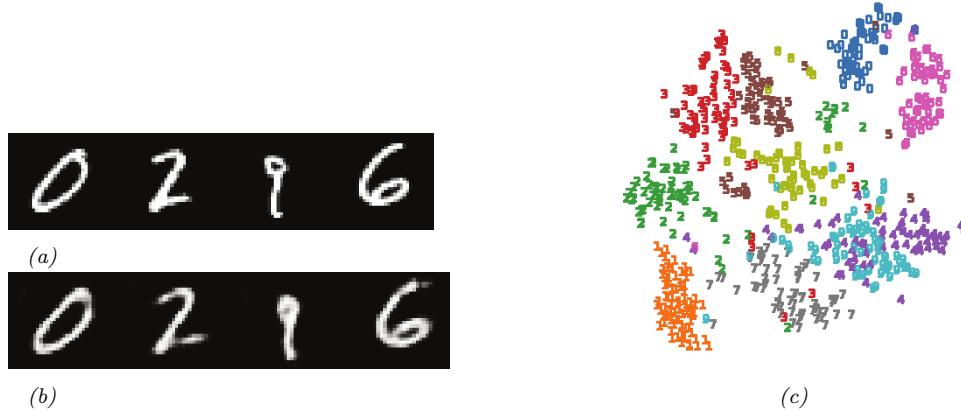


Figure 16.14: (a) Some MNIST digits. (b) Reconstruction of these images using a convolutional autoencoder. (c) t-SNE visualization of the 20-d embeddings. The colors correspond to class labels, which were not used during training. Generated by `ae_mnist_conv_jax.ipynb`.

16.3.3 Autoencoders

An **autoencoder** is a neural network that maps inputs \mathbf{x} to a low-dimensional latent space using an **encoder**, $\mathbf{z} = f_e(\mathbf{x})$, and then attempts to reconstruct the inputs using a **decoder**, $\hat{\mathbf{x}} = f_d(\mathbf{z})$. The model is trained to minimize

$$\mathcal{L}(\boldsymbol{\theta}) = \|\mathbf{r}(\mathbf{x}) - \mathbf{x}\|_2^2 \quad (16.23)$$

where $\mathbf{r}(\mathbf{x}) = f_d(f_e(\mathbf{x}))$. (We can also replace squared error with more general conditional log likelihoods.) See Figure 16.13 for an illustration of a 3 layer AE.

For image data, we can make the encoder be a convolutional network, and the decoder be a transpose convolutional network. We can use this to compute low dimensional embeddings of image

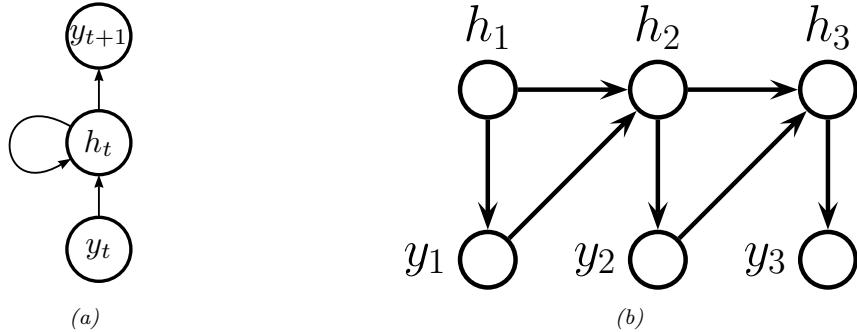


Figure 16.15: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

data. For example, suppose we fit such a model to some MNIST digits. We show the reconstruction abilities of such a model in Figure 16.14b. In Figure 16.14c, we show a 2d visualization of the 20-dimensional embedding space computed using t-SNE. The colors correspond to class labels, which were not used during training. We see fairly good separation, showing that images which are visually similar are placed close to each other in the embedding space, as desired. (See also Section 21.2.3, where we compare AEs with variational AEs.)

16.3.4 Recurrent neural networks (RNNs)

A **recurrent neural network (RNN)** is a network with a recurrent layer, as in Equation (16.15). This is illustrated in Figure 16.15. Formally this defines the following probability distribution over sequences:

$$p(\mathbf{y}_{1:T}) = \sum_{\mathbf{h}_{1:T}} p(\mathbf{y}_{1:T}, \mathbf{h}_{1:T}) = \sum_{\mathbf{h}_{1:T}} \mathbb{I}(\mathbf{h}_1 = \mathbf{h}_1^*) p(\mathbf{y}_1 | \mathbf{h}_1) \prod_{t=2}^T p(\mathbf{y}_t | \mathbf{h}_t) \mathbb{I}(\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1})) \quad (16.24)$$

where \mathbf{h}_t is the deterministic hidden state, computed from the last hidden state and last output using $f(\mathbf{h}_{t-1}, \mathbf{y}_{t-1})$. (At training time, \mathbf{y}_{t-1} is observed, but at prediction time, it is generated.)

In a vanilla RNN, the function f is a simple MLP. However, we can also use attention to selectively update parts of the state vector based on similarity between the input the previous state, as in the **GRU** (gated recurrent unit) model, and the **LSTM** (long short term memory) model. We can also make the model into a conditional sequence model, by feeding in extra inputs to the f function. See e.g., [Mur22, Ch. 15] for more details on RNNs.

16.3.5 Transformers

Consider the problem of classifying each word in a sentence, for example with its part of speech tag (noun, verb, etc). That is, we want to learn a mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X} = \mathcal{V}^T$ is the set of input sequences defined over (word) vocabulary \mathcal{V} , T is the length of the sentence, and $\mathcal{Y} = \mathcal{T}^T$ is the set of output sequences, defined over (tag) vocabulary \mathcal{T} . To do well at this task, we need to learn a contextual embedding of each word. RNNs process one token at a time, so the embedding of

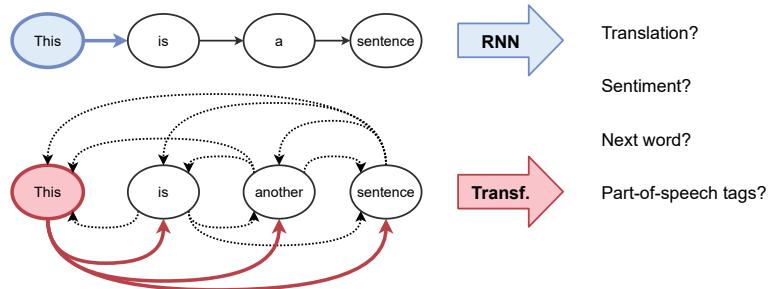


Figure 16.16: Visualizing the difference between an RNN and a transformer. From [Jos20]. Used with kind permission of Chaitanya Joshi.

the word at location t , \mathbf{z}_t , depends on the hidden state of the network, \mathbf{s}_t , which may be a lossy summary of all the previously seen words. We can create bidirectional RNNs so that future words can also affect the embedding of \mathbf{z}_t , but this dependence is still mediated via the hidden state. An alternative approach is to compute \mathbf{z}_t as a direct function of all the other words in the sentence, by using the attention operator discussed in Section 16.2.7 rather than using hidden state. This is called an (encoder-only) **transformer**, and is used by models such as BERT [Dev+19]. This idea is sketched in Figure 16.16.

It is also possible to create a decoder-only transformer, in which each output \mathbf{y}_t only attends to all the previously generated outputs, $\mathbf{y}_{1:t-1}$. This can be implemented using masked attention, and is useful for generative language models, such as GPT (see Section 22.4.1). We can combine the encoder and decoder to create a conditional sequence-to-sequence model, $p(\mathbf{y}_{1:T_y}|\mathbf{x}_{1:T_x})$, as proposed in the original transformer paper [Vas+17c]. See Supplementary Section 16.1.1 and [PH22] for more details.

It has been found that large transformers are very flexible sequence-to-sequence function approximators, if trained on enough data (see e.g., [Lin+21a] for a review in the context of NLP, and [Kha+21; Han+20; Zan21] for reviews in the context of computer vision). The reasons why they work so well are still not very clear. However, some initial insights can be found in, e.g., [Rag+21; WGY21; Nel21; BP21]. See also Supplementary Section 16.1.2.5 where we discuss the connection with graph neural networks.

16.3.6 Graph neural networks (GNNs)

It is possible to define neural networks for working with graph-structured data. These are called **graph neural networks** or **GNNs**. See Supplementary Section 16.1.2 for details.

17 Bayesian neural networks

This chapter is coauthored with Andrew Wilson.

17.1 Introduction

Deep neural networks (DNNs) are usually trained using a (penalized) maximum likelihood objective to find a single setting of parameters. However, large flexible models like neural networks can represent many functions, corresponding to different parameter settings, which fit the training data well, yet generalize in different ways. (This phenomenon is known as **underspecification** (see e.g., [D'A+20]; see Figure 17.11 for an illustration.) Considering all of these different models together can lead to improved accuracy and uncertainty representation. This can be done by computing the posterior predictive distribution using Bayesian model averaging:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \quad (17.1)$$

where $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})$.

The main challenges in applying Bayesian inference to DNNs are specifying suitable priors, and efficiently computing the posterior, which is challenging due to the large number of parameters and the large datasets. The application of Bayesian inference to DNNs is sometimes called **Bayesian deep learning** or **BDL**. By contrast, the term **deep Bayesian learning** or **DBL** refers to the use of deep models to help speed up Bayesian inference of “classical” models, usually by training amortized inference networks that can be used as part of a variational inference or importance sampling algorithm, as discussed in Section 10.1.5.) For more details on the topic of BDL, see e.g., [PS17; Wil20; WI20; Jos+22; Kha20].

17.2 Priors for BNNs

To perform Bayesian inference for the parameters of a DNN, we need to specify a prior $p(\boldsymbol{\theta})$. [Nal18; WI20; For22] discusses the issue of prior selection at length. Here we just give a brief summary of common approaches.

17.2.1 Gaussian priors

Consider an MLP with one hidden layer with activation function φ and a linear output:

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_2 \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \quad (17.2)$$

(If the output is nonlinear, such as a softmax transform, we can fold it into the loss function during training.) If we have two hidden layers this becomes

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_3 (\varphi(\mathbf{W}_2 \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)) + \mathbf{b}_3 \quad (17.3)$$

In general, with $L - 1$ hidden layers and a linear output, we have

$$f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}_L (\cdots \varphi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_L \quad (17.4)$$

We need to specify the priors for \mathbf{W}_l and \mathbf{b}_l for $l = 1 : L$. The most common choice is to use a factored Gaussian prior:

$$\mathbf{W}_\ell \sim \mathcal{N}(\mathbf{0}, \alpha_\ell^2 \mathbf{I}), \mathbf{b}_\ell \sim \mathcal{N}(\mathbf{0}, \beta_\ell^2 \mathbf{I}) \quad (17.5)$$

The **Xavier initialization** or **Glorot initialization**, named after the first author of [GB10], is to set

$$\alpha_\ell^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \quad (17.6)$$

where n_{in} is the fan-in of a node in level ℓ (number of weights coming into a neuron), and n_{out} is the fan-out (number of weights going out of a neuron). **LeCun initialization**, named after Yann LeCun, corresponds to using

$$\alpha_\ell^2 = \frac{1}{n_{\text{in}}} \quad (17.7)$$

We can get a better understanding of these priors by considering the effect they have on the corresponding distribution over functions that they define. To help understand this correspondence, let us reparameterize the model as follows:

$$\mathbf{W}_\ell = \alpha_\ell \boldsymbol{\eta}_\ell, \boldsymbol{\eta}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \mathbf{b}_\ell = \beta_\ell \boldsymbol{\epsilon}_\ell, \boldsymbol{\epsilon}_\ell \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (17.8)$$

Hence every setting of the prior hyperparameters specifies the following random function:

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \alpha_L \boldsymbol{\eta}_L (\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x} + \beta_1 \boldsymbol{\epsilon}_1)) + \beta_L \boldsymbol{\epsilon}_L \quad (17.9)$$

To get a feeling for the effect of these hyperparameters, we can sample MLP parameters from this prior and plot the resulting random functions. We use a sigmoid nonlinearity, so $\varphi(a) = \sigma(a)$. We consider $L = 2$ layers, so \mathbf{W}_1 are the input-to-hidden weights, and \mathbf{W}_2 are the hidden-to-output weights. We assume the input and output are scalars, so we are generating random nonlinear 1d mappings $f : \mathbb{R} \rightarrow \mathbb{R}$.

Figure 17.1(a) shows some sampled functions where $\alpha_1 = 5$, $\beta_1 = 1$, $\alpha_2 = 1$, $\beta_2 = 1$. In Figure 17.1(b) we increase α_1 ; this allows the first layer weights to get bigger, making the sigmoid-like

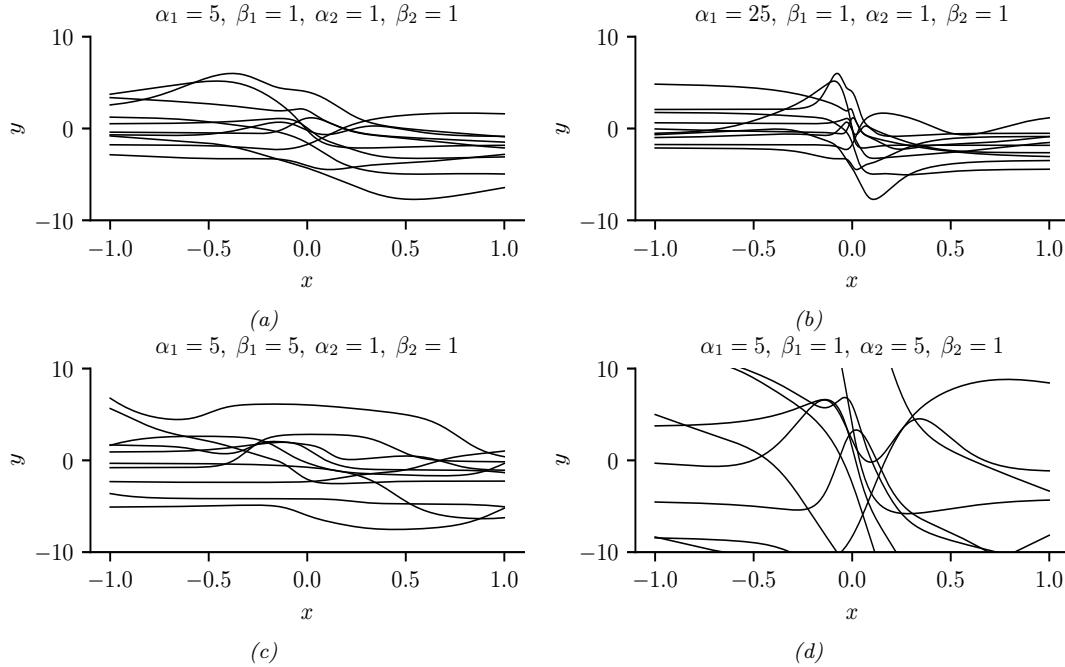


Figure 17.1: The effects of changing the hyperparameters on an MLP with one hidden layer. (a) Random functions sampled from a Gaussian prior with hyperparameters $\alpha_1 = 5, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$. (b) Increasing α_1 by a factor of 5. (c) Increasing β_1 by a factor of 5. (d) Increasing α_2 by a factor of 5. Generated by [mlp_priors_demo.ipynb](#).

shape of the functions steeper. In Figure 17.1(c), we increase β_1 ; this allows the first layer biases to get bigger, which allows the center of the sigmoid to shift left and right more, away from the origin. In Figure 17.1(d), we increase α_2 ; this allows the second layer linear weights to get bigger, making the functions more “wiggly” (greater sensitivity to change in the input, and hence larger dynamic range).

The above results are specific to the case of sigmoidal activation functions. ReLU units can behave differently. For example, [WI20, App. E] show that for MLPs with ReLU units, if we set $\beta_\ell = 0$, so the bias terms are all zero, the effect of changing α_ℓ is just to rescale the output. To see this, note that Equation (17.9) simplifies to

$$f(\mathbf{x}; \boldsymbol{\alpha}, \boldsymbol{\beta} = \mathbf{0}) = \alpha_L \boldsymbol{\eta}_L(\cdots \varphi(\alpha_1 \boldsymbol{\eta}_1 \mathbf{x})) = \alpha_L \cdots \alpha_1 \boldsymbol{\eta}_L(\cdots \varphi(\boldsymbol{\eta}_1 \mathbf{x})) \quad (17.10)$$

$$= \alpha_L \cdots \alpha_1 f(\mathbf{x}; (\boldsymbol{\alpha} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0})) \quad (17.11)$$

where we used the fact that for ReLU, $\varphi(\alpha z) = \alpha \varphi(z)$ for any positive α , and $\varphi(\alpha z) = 0$ for any negative α (since the preactivation $z \geq 0$). In general, it is the ratio of α and β that matters for determining what happens to input signals as they propagate forwards and backwards through a randomly initialized model; for details, see e.g., [Bah+20].

We see that initializing the model’s parameters at a particular random value is like sampling a

point from this prior over functions. In the limit of infinitely wide neural networks, we can derive this prior distribution analytically: this is known as a **neural network Gaussian process**, and is explained in Section 18.7.

17.2.2 Sparsity-promoting priors

Although Gaussian priors are simple and widely used, they are not the only option. For some applications, it is useful to use **sparsity promoting priors**, such as the Laplace, which encourage most of the weights (or channels in a CNN) to be zero (cf. Section 15.2.6). For details, see [Hoe+21].

17.2.3 Learning the prior

We have seen how different priors for the parameters correspond to different priors over functions. We could in principle set the hyperparameters (e.g., the α and β parameters of the Gaussian prior) using grid search to optimize cross-validation loss. However, cross-validation can be slow, particularly if we allow different priors for each layer of the network, as our grid search will grow exponentially with the number of hyperparameters we wish to determine.

An alternative is to use gradient based methods to optimize the marginal likelihood

$$\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta}) = \int \log p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\alpha}, \boldsymbol{\beta})d\boldsymbol{\theta} \quad (17.12)$$

This approach is known as empirical Bayes (Section 3.7) or **evidence maximization**, since $\log p(\mathcal{D}|\boldsymbol{\alpha}, \boldsymbol{\beta})$ is also called the evidence [Mac92a; WS93; Mac99]. This can give rise to sparse models, as we discussed in the context of automatic relevancy determination (Section 15.2.8). Unfortunately, computing the marginal likelihood is computationally difficult for large neural networks.

Learning the prior is more meaningful if we can do it on a separate, but related dataset. In [SZ+22] they propose to train a model on an initial, large dataset \mathcal{D}_1 (possibly unsupervised) to get a point estimate, $\hat{\boldsymbol{\theta}}_1$, from which they can derive an approximate low-rank Gaussian posterior, using the SWAG method (Section 17.3.8). They then use this informative prior when fine-tuning the model on a downstream dataset \mathcal{D}_2 . The fine-tuning can either be a MAP estimate $\hat{\boldsymbol{\theta}}_2$ or some approximate posterior, $p(\boldsymbol{\theta}_2|\mathcal{D}_2, \mathcal{D}_1)$, e.g., computed using MCMC (Section 17.3.7). They call this technique “**Bayesian transfer learning**”. (See Section 19.5.1 for more details on transfer learning.)

17.2.4 Priors in function space

Typically, the relationship between the prior distribution over parameters and the functions preferred by the prior is not transparent. In some cases, it can be possible to pick more informative priors based on principles such as desired invariances that we want the function to satisfy (see e.g., [Nal18]). [FBW21] introduces *residual pathway priors*, providing a mechanism for encoding high level concepts into prior distributions, such as locality, independencies, and symmetries, without constraining model flexibility. A different approach to encoding interpretable priors over functions leverages kernel methods such as Gaussian processes (e.g., [Sun+19a]), as we discuss in Section 18.1.

17.2.5 Architectural priors

Beyond specifying the parametric prior, it is important to note that the architecture of the model can have an even larger effect on the induced distribution over functions, as argued in Wilson and Izmailov [WI20] and Izmailov et al. [Izm+21b]. For example, a CNN architecture encodes prior knowledge about translation equivariance, due to its use of convolution, and hierarchical structure, due to its use of multiple layers. Other forms of inductive bias are induced by different architectures, such as RNNs. (Models such as transformers have weaker inductive bias, but consequently often need more data to perform well.) Thus we can think of the field of **neural architecture search** (reviewed in [EMH19]) as a form of structural prior learning.

In fact, with a suitable architecture, we can often get good results using random (untrained) models. For example, Ulyanov, Vedaldi, and Lempitsky [UVL18] showed that an untrained CNN with random parameters (sampled from a Gaussian) often works very well for low-level image processing tasks, such as image denoising, super-resolution, and image inpainting. The resulting prior over functions has been called the **deep image prior**. Similarly, Pinto and Cox [PC12] showed that untrained CNNs with the right structure can do well at face recognition. Moreover, Zhang et al. [Zha+17] show that randomly initialized CNNs can process data to provide features that greatly improve the performance of other models, such as kernel methods.

17.3 Posteriors for BNNs

There are a large number of different approximate inference schemes that have been applied to Bayesian neural networks, with different strengths and limitations. In the sections below, we briefly describe some of these.

17.3.1 Monte Carlo dropout

Monte Carlo dropout (MCD) [GG16; KG17] is a very simple and widely used method for approximating the Bayesian predictive distribution. Usually stochastic dropout layers are added as a form of regularization, and are “turned off” at test time, as described in Section 16.2.6. However, the idea in MCD is to also perform random sampling at test time. More precisely, we drop out each hidden unit by sampling from a Bernoulli(p) distribution; we repeat this procedure S times, to create S distinct models. We then create an equally weighted average of the predictive distributions for each of these models:

$$p(y|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(y|\mathbf{x}, \boldsymbol{\theta}^s) \quad (17.13)$$

where $\boldsymbol{\theta}^s$ is a version of the MAP parameter estimate where we randomly drop out some connections.

We give an example of this process in action in Figure 17.2. We see that it successfully captures uncertainty due to “out of distribution” inputs. (See Section 19.3.2 for more discussion of OOD detection.)

One drawback of MCD is that it is slow at test time. However this can be overcome by “distilling” the model’s predictions into a deterministic “student” network, as we discuss in Section 17.3.10.3.

A more fundamental problem is that MCD does not give proper uncertainty estimates, as argued in [Osb16; LF+21]. The problem is the following. Although MCD can be viewed as a form of variational

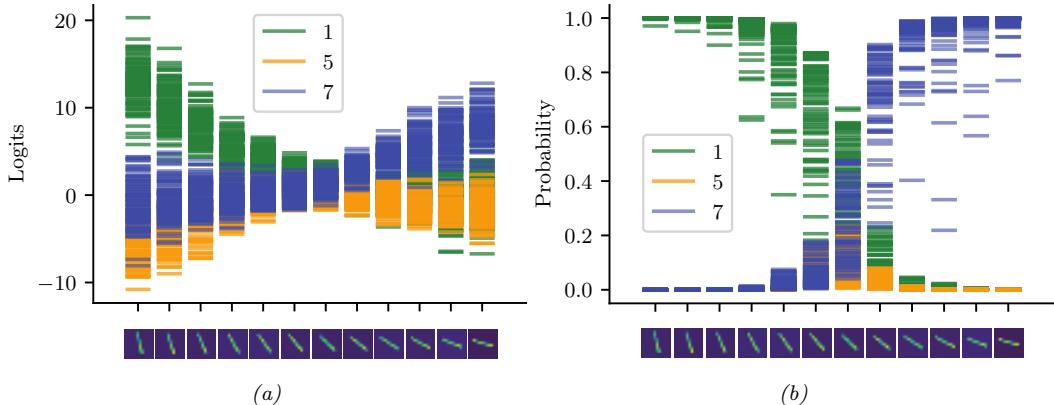


Figure 17.2: Illustration of MC dropout applied to the LeNet architecture. The inputs are some rotated images of the digit 1 from the MNIST dataset. (a) Softmax inputs (logits). (b) Softmax outputs (probabilities). We see that the inputs are classified as digit 7 for the last three images (as shown by the probabilities), even though the model has high uncertainty (as shown by the logits). Adapted from Figure 4 of [GG16]. Generated by `mnist_classification_mc_dropout.ipynb`

inference [GG16], this is only true under a degenerate posterior approximation, corresponding to a mixture of two delta functions, one at 0 (for dropped out nodes) and one at the MLE. This posterior will not converge to the true posterior (which is a delta function at the MLE) even as the training set size goes to infinity, since we are always dropping out hidden nodes with a constant probability p [Osb16]. Fortunately this pathology can be fixed if the noise rate is optimized [GHK17]. For more details, see e.g., [HGMG18; NHLS19; LF+21].

17.3.2 Laplace approximation

In Section 7.4.3, we introduced the Laplace approximation, which computes a Gaussian approximation to the posterior, $p(\boldsymbol{\theta}|\mathcal{D})$, centered at the MAP estimate, $\boldsymbol{\theta}^*$. The posterior prediction matrix is equal to the Hessian of the negative log joint computed at the mode. The benefits of this approach are that it is simple, and it can be used to derive a Bayesian estimate from a pretrained model. The main disadvantage is that computing the Hessian can be expensive. In addition, it may not be positive definite, since the log likelihood of DNNs is non-convex. It is therefore common to use a Gauss-newton approximation to the Hessian instead, as we explain below.

Following the notation of [Dax+21], let $\mathbf{f}(\mathbf{x}_n, \boldsymbol{\theta}) \in \mathbb{R}^C$ be the prediction function with C outputs, and $\boldsymbol{\theta} \in \mathbb{R}^P$ be the parameter vector. Let $\mathbf{r}(\mathbf{y}; \mathbf{f}) = \nabla_{\mathbf{f}} \log p(\mathbf{y}|\mathbf{f})$ be the residual¹, and $\Lambda(\mathbf{y}; \mathbf{f}) = -\nabla_{\mathbf{f}}^2 \log p(\mathbf{y}|\mathbf{f})$ be the per-input noise term. In addition, let $\mathbf{J} \in \mathbb{R}^{C \times P}$ be the Jacobian, $[\mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})]_{ci} = \frac{\partial f_c(\mathbf{x}, \boldsymbol{\theta})}{\partial \theta_i}$, and $\mathbf{H} \in \mathbb{R}^{C \times P \times P}$ be the Hessian, $[\mathbf{H}_{\boldsymbol{\theta}}(\mathbf{x})]_{cij} = \frac{\partial^2 f_c(\mathbf{x}, \boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}$. Then the gradient and Hessian

1. In the Gaussian case, this term becomes $\nabla_{\mathbf{f}} \|\mathbf{y} - \mathbf{f}\|^2 = 2\|\mathbf{y} - \mathbf{f}\|$, so it can be interpreted as a residual error.

of the log likelihood are given by the following [IKB21]:

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) \quad (17.14)$$

$$\nabla_{\boldsymbol{\theta}}^2 \log p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \mathbf{H}_{\boldsymbol{\theta}}(\mathbf{x})^T \mathbf{r}(\mathbf{y}; \mathbf{f}) - \mathbf{J}_{\boldsymbol{\theta}}(\mathbf{x})^T \boldsymbol{\Lambda}(\mathbf{y}; \mathbf{f}) \mathbf{J}_{\boldsymbol{\theta}}(\boldsymbol{\theta}) \quad (17.15)$$

Since the network Hessian \mathbf{H} is usually intractable to compute, it is usually dropped, leaving only the Jacobian term. This is called the **generalized Gauss-Newton** or **GGN** approximation [Sch02; Mar20]. The GGN approximation is guaranteed to be positive definite. By contrast, this is not true for the original Hessian in Equation (17.15), since the objective is not convex. Furthermore, computing the Jacobian term is cheaper to compute than the Hessian.

Putting it all together, for a Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{m}_0, \mathbf{S}_0)$, the Laplace approximation becomes $p(\boldsymbol{\theta}|\mathcal{D}) \approx (\mathcal{N}|\boldsymbol{\theta}^*, \boldsymbol{\Sigma}_{\text{GGN}})$, where

$$\boldsymbol{\Sigma}_{\text{GGN}}^{-1} = \sum_{n=1}^N \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n)^T \boldsymbol{\Lambda}(\mathbf{y}_n; \mathbf{f}_n) \mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}_n) + \mathbf{S}_0^{-1} \quad (17.16)$$

Unfortunately inverting this matrix takes $O(P^3)$ time, so for models with many parameters, further approximations are usually used. The simplest is to use a diagonal approximation, which takes $O(P)$ time and space. A more sophisticated approach is presented in [RBB18a], which leverages the **KFAC** (Kronecker factored curvature) approximation of [MG15]. This approximates the covariance of each layer using a Kronecker product.

A limitation of the Laplace approximation is that the posterior covariance is derived from the Hessian evaluated at the MAP parameters. This means Laplace forms a highly *local* approximation: even if the non-Gaussian posterior could be well-described by a Gaussian distribution, the Gaussian distribution *formed using Laplace* only captures the local characteristics of the posterior at the MAP parameters — and may therefore suffer badly from local optima, providing overly compact or diffuse representations. In addition, the curvature information is only used after the model has been estimated, and not during the model optimization process. By contrast, variational inference (Section 17.3.3) can provide more accurate approximations for comparable cost.

17.3.3 Variational inference

In fixed-form variational inference (Section 10.2), we choose a distribution for the posterior approximation $q_{\psi}(\boldsymbol{\theta})$ and minimize $D_{\text{KL}}(q \parallel p)$, with respect to ψ . We often choose a Gaussian approximate posterior, $q_{\psi}(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, which lets us use the reparameterization trick to create a low variance estimator of the gradient of the ELBO (see Section 10.2.1). Despite the use of a Gaussian, the parameters that minimize the KL objective are often different what we would find with the Laplace approximation (Section 17.3.2).

Variational methods for neural networks date back to at least Hinton and Camp [HC93]. In deep learning, [Gra11] revisited variational methods, using a Gaussian approximation with a diagonal covariance matrix. This approximates the distribution of every parameter in the model by a univariate Gaussian, where the mean is the point estimate, and the variance captures the uncertainty, as shown in Figure 17.3. This approach was improved further in [Blu+15], who used the reparameterization trick to compute lower variance estimates of the ELBO; they called their method **Bayes by backprop** (**BBB**). This is essentially identical to the SVI algorithm in Algorithm 10.2, except the likelihood becomes $p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})$ from the DNN, and the prior $p_{\xi}(\boldsymbol{\theta})$ and variational posterior $q_{\psi}(\boldsymbol{\theta})$ are Gaussians.

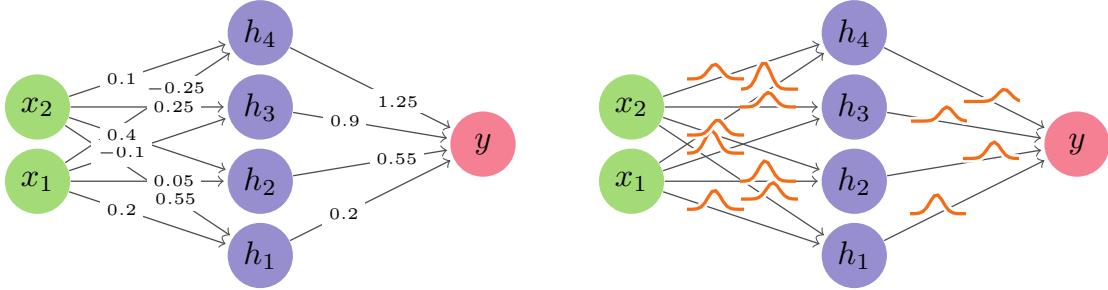


Figure 17.3: Illustration of an MLP with (left) a point estimate for each weight, (right) a marginal distribution for each weight, corresponding to a fully factored posterior approximation.

Many extensions of the BBB have been proposed. In [KSW15], they propose the **local reparameterization trick**, that samples the activations $\mathbf{a} = \mathbf{W}\mathbf{z}$ at each layer, instead of the weights \mathbf{W} , which results in a lower variance estimate of the ELBO gradient. In [Osa+19a], they used the **variational online Gauss-Newton (VOGN)** method of [Kha+18], for improved scalability. VOGN is a noisy version of natural gradient descent, where the extra noise emulates the effect of variational inference. In [Mis+18], they replaced the diagonal approximation with a low-rank plus diagonal approximation, and used VOGN for fitting. In [Tra+20b], they use a rank-one plus diagonal approximation known as **NAGVAC** (see Section 10.2.1.3). In this case, there are only 3 times as many parameters as when computing a point estimate (for the variational mean, variance, and rank-one vector), making the approach very scalable. In addition, in this case it is possible to analytically compute the natural gradient, which speeds up model fitting (see Section 6.4). Many other variational methods have also been proposed (see e.g., [LW16; Zha+18; Wu+19a; HHK19]). See also Section 17.5.4 for a discussion of online VI for DNNs.

17.3.4 Expectation propagation

Expectation propagation (EP) is similar to variational inference, except it locally optimizes $D_{\text{KL}}(p \parallel q)$ instead of $D_{\text{KL}}(q \parallel p)$, where p is the exact posterior and q is the approximate posterior. For details, see Section 10.7.

A special case of EP is the assumed density filtering (ADF) algorithm of Section 8.6, which is equivalent to the first pass of ADF. In Section 8.6.3 we show how to apply ADF to online logistic regression. In [HLA15a], they extend ADF to the case of BNNs; they called their method probabilistic backpropagation or **PBP**. They approximate every parameter in the model by a Gaussian factor, as in Figure 17.3. See Section 17.5.3 for the details.

17.3.5 Last layer methods

A very simple approximation to the posterior is to only “be Bayesian” about the weights in the final layer, and to use MAP estimates for all the other parameters. This is called the **neural-linear** approximation [RTS18]. In more detail, let $\mathbf{z} = f(\mathbf{x}; \boldsymbol{\theta})$ be the predicted outputs (e.g., logits) of the model before any optional final nonlinearity. We assume this has the form $\mathbf{z} = \mathbf{w}_L^T \phi(\mathbf{x}; \boldsymbol{\theta})$,

where $\phi(\mathbf{x})$ are the features extracted by the first $L - 1$ layers. This gives us a Bayesian GLM. We can use standard techniques, such as the Laplace approximation (Section 15.3.5), to compute $p(\mathbf{w}_L | \mathcal{D}) = \mathcal{N}(\boldsymbol{\mu}_L, \boldsymbol{\Sigma}_L)$, given $\phi()$. To estimate the parameters of the feature extractor, we can optimize the log-likelihood in the usual way. Given the posterior over the last layer weights, we can compute the posterior predictive distribution over the logits using

$$p(\mathbf{z} | \mathbf{x}, \mathcal{D}) = \mathcal{N}(\mathbf{z} | \boldsymbol{\mu}_L \phi(\mathbf{x}), \phi(\mathbf{x}) \boldsymbol{\Sigma}_L \phi(\mathbf{x})^\top) \quad (17.17)$$

This can be passed through the final softmax layer to compute $p(\mathbf{y} | \mathbf{x}, \mathcal{D})$ as described in Section 15.3.6.

In [KHH20] they show this can reduce overconfidence in predictions for inputs that are far from the training data. However, this approach ignores uncertainty introduced by the earlier feature extraction layers, where most of the parameters reside. We discuss a solution to this in Section 17.3.6.

17.3.6 SNGP

It is possible to combine DNNs with Gaussian process (GP) models (Chapter 18), by using the DNN to act as a feature extractor, which is then fed into the kernel in the final layer. This is called “deep kernel learning” (see Section 18.6.6).

One problem with this is that the feature extractor may lose information which is not needed for classification accuracy, but which is needed for robust performance on out-of-distribution inputs (see Section 17.4.6.2). The basic problem is that, in a classification problem, there is no reduction in training accuracy (log likelihood) if points which are far away are projected close together, as long as they are on the correct side of the decision boundary. Thus the distances between two inputs can be erased by the feature extraction layers, so that OOD inputs appear to the final layer to be close to the training set.

One solution to this is to use the **SNGP** (spectrally normalized Gaussian process) method of [Liu+20d; Liu+22a]. This constrains the feature extraction layers to be “distance preserving”, so that two inputs that are far apart in input space remain far apart after many layers of feature extraction, by using spectral normalization of the weights to bound the Lipschitz constant of the feature extractor. The overall approach ensures that information that is relevant for computing the confidence of a prediction, but which might be irrelevant to computing the label of a prediction, is not lost. This can help performance in tasks such as out-of-distribution detection (Section 17.4.6.2).

17.3.7 MCMC methods

Some of the earliest work on inference for BNNs was done by Radford Neal, who proposed to use Hamiltonian Monte Carlo (Section 12.5) to approximate the posterior [Nea96]. This is generally considered the gold standard method, since it does not make strong assumptions about the form of the posterior. For more recent work on scaling up HMC for BNNs, see e.g., [Izm+21b; CJ21].

We give a simple example of vanilla HMC in Figure 17.4, where we fit a shallow MLP to a small 2d binary dataset. We plot the mean and standard deviation of the posterior predictive distribution, $p(y = 1 | \mathbf{x}; \mathcal{D})$. We see that the uncertainty is higher as we move away from the training data. (Compare to Bayesian logistic regression in 1d in Figure 15.8a.)

However, a significant limitation of standard MCMC procedures, including HMC, is that they require access to the full training set at each step. Stochastic gradient MCMC methods, such as

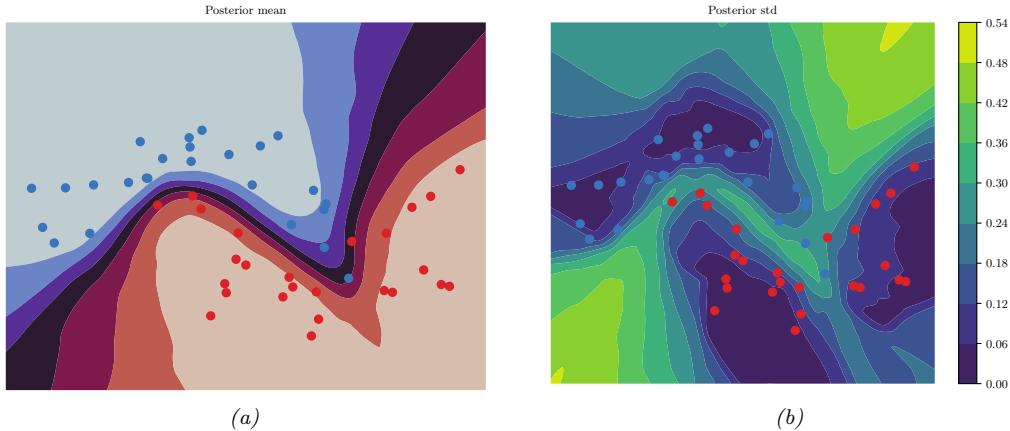


Figure 17.4: Illustration of an MLP fit to the two-moons dataset using HMC. (a) Posterior mean. (b) Posterior standard derivation. The uncertainty increases as we move away from the training data. Generated by [bnn_mlp_2d_hmc.ipynb](#).

SGLD, operate instead using mini-batches of data, offering a scalable alternative, as we discuss in Section 12.7.1. For an example of SGLD applied to an MLP, see Section 19.3.3.1.

17.3.8 Methods based on the SGD trajectory

In [MHB17; SL18; CS18], it was shown that, under some assumptions, the iterates produced by stochastic gradient descent (SGD), when run at a fixed learning rate, correspond to samples from a Gaussian approximation to the posterior centered at a local mode, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}, \boldsymbol{\Sigma})$. We can therefore use SGD to generate approximate posterior samples. This is similar to SG-MCMC methods, except we do not add explicit gradient noise, and the learning rate is held constant.

In [Izm+18], they noted that these SGD solutions (with fixed learning rate) surround the periphery of points of good generalization, as shown in Figure 17.5. This is in part because SGD does not converge to a local optimum unless the learning rate is annealed to 0. They therefore proposed to compute the average of several SGD samples, each one collected after a certain interval (e.g., one epoch of training), to get $\bar{\boldsymbol{\theta}} = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s$. They call this **stochastic weight averaging (SWA)**. They showed that the resulting point tends to correspond to a broader local minimum than the SGD solutions (see Figure 17.10), resulting in better generalization performance.

The SWA approach is related to Polyak-Ruppert averaging, which is often used in convex optimization. The difference is that Polyak-Ruppert typically assumes the learning rate decays to zero, and uses an exponential moving average (EMA) of iterates, rather than an equal average; Polyak-Ruppert averaging is mainly used to reduce variance in the SGD estimate, rather than as a method to find points of better generalization.

The SWA approach is also related to **snapshot ensembles** [Hua+17a], and **fast geometric ensembles** [Gar+18c]; these methods save the parameters $\boldsymbol{\theta}_s$ after increasing and decreasing the learning rate multiple times in a cyclical fashion, and then computing the *average of the predictions* using $p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_s)$, rather than computing the *average of the parameters* and

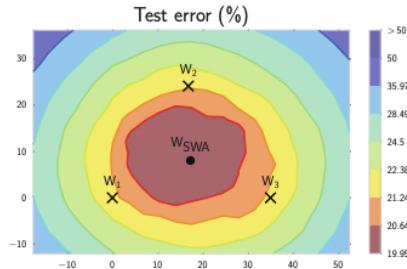


Figure 17.5: Illustration of stochastic weight averaging (SWA). The three crosses represent different SGD solutions. The star in the middle is the average of these parameter values. From Figure 1 of [Izm+18]. Used with kind permission of Andrew Wilson.

predicting with a single model (which is faster). Moreover, by finding a flat region, representing a “center or mass” in the posterior, SWA can be seen as approximating the Bayesian model average in Equation 17.1 with a single model.

In [Mad+19], they proposed to fit a Gaussian distribution to the set of samples produced by SGD near a local mode. They use the SWA solution as the mean of the Gaussian. For the covariance matrix, they use a low-rank plus diagonal approximation of the form $p(\boldsymbol{\theta}|\mathcal{D}) = \mathcal{N}(\boldsymbol{\theta}|\bar{\boldsymbol{\theta}}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_{\text{diag}} + \boldsymbol{\Sigma}_{\text{lr}})/2$, $\boldsymbol{\Sigma}_{\text{diag}} = \text{diag}(\bar{\boldsymbol{\theta}}^2 - (\bar{\boldsymbol{\theta}})^2)$, $\bar{\boldsymbol{\theta}} = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s$, $\bar{\boldsymbol{\theta}}^2 = \frac{1}{S} \sum_{s=1}^S \boldsymbol{\theta}_s^2$, and $\boldsymbol{\Sigma}_{\text{lr}} = \frac{1}{S} \boldsymbol{\Delta} \boldsymbol{\Delta}^\top$ is the sample covariance matrix of the last K samples of $\boldsymbol{\Delta}_i = (\boldsymbol{\theta}_i - \bar{\boldsymbol{\theta}}_i)$, where $\bar{\boldsymbol{\theta}}_i$ is the running average of the parameters from the first i samples. They call this method **SWAG**, which stands for “stochastic weight averaging with Gaussian posterior”. This can be used to generate an arbitrary number of posterior samples at prediction time. They show that SWAG scales to large residual networks with millions of parameters, and large datasets such as ImageNet, with improved accuracy and calibration over conventional SGD training, and no additional training overhead.

17.3.9 Deep ensembles

Many conventional approximate inference methods focus on approximating the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ in a local neighborhood around one of the posterior modes. While this is often not a major limitation in classical machine learning, modern deep neural networks have highly multi-modal posteriors, with parameters in different modes giving rise to very different functions. On the other hand, the functions in a neighborhood of a single mode may make fairly similar predictions. So using such a local approximation to compute the posterior predictive will underestimate uncertainty and generalize more poorly.

A simple alternative method is to train multiple models, and then to approximate the posterior using an equally weighted mixture of delta functions,

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}_m) \quad (17.18)$$

where M is the number of models, and $\hat{\boldsymbol{\theta}}_m$ is the MAP estimate for model m . See Figure 17.6 for a

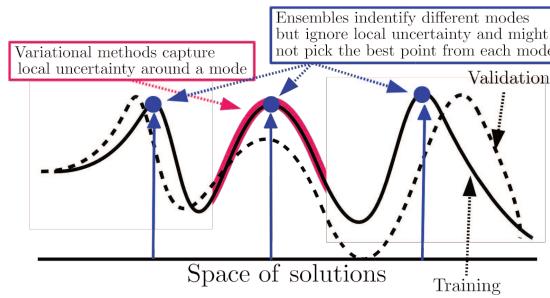


Figure 17.6: Cartoon illustration of the NLL as it varies across the parameter space. Subspace methods (red) model the local neighborhood around a local mode, whereas ensemble methods (blue) approximate the posterior using a set of distinct modes. From Figure 1 of [FHL19]. Used with kind permission of Balaji Lakshminarayanan.

sketch. This approach is called **deep ensembles** [LPB17; FHL19].

The models can differ in terms of their random seed used for initialization [LPB17], or hyperparameters [Wen+20c], or architecture [Zai+20], or all of the above. In addition, [DF21; TB22] discusses how to add an explicit repulsive term to ensure functional diversity between the ensemble members. This way, each member corresponds to a distinct prediction function. Combining these is more effective than combining multiple samples from the same basin of attraction, especially in the presence of dataset shift [Ova+19].

17.3.9.1 Multi-SWAG

We can further improve on this approach by fitting a Gaussian to each local mode using the SWAG method from Section 17.3.8 to get a mixture of Gaussians approximation:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M \mathcal{N}(\boldsymbol{\theta}|\hat{\boldsymbol{\theta}}_m, \boldsymbol{\Sigma}_m) \quad (17.19)$$

This approach is known as **MultiSWAG** [WI20]. MultiSWAG performs a Bayesian model average both across multiple basins of attraction, like deep ensembles, but also within each basin, and provides an easy way to generate an arbitrary number of posterior samples, $S > M$, in an any-time fashion.

17.3.9.2 Deep ensembles with random priors

The standard way to fit each member of a deep ensemble is to initialize them each with a different random set of parameters, but train them all on the same data. Unfortunately this can result in the predictions from each ensemble member being rather similar, which reduces the benefit of the approach. One way to increase diversity is to train each member on a different subset of the data; this is called **bootstrap sampling**. Another approach is to define the i 'th ensemble member $g_i(\mathbf{x})$ to be the addition of a trainable model $t_i(\mathbf{x})$ and a fixed, but random, **prior network**, $p_i(\mathbf{x})$, to get

$$g_i(\mathbf{x}; \boldsymbol{\theta}_i) = t_i(\mathbf{x}; \boldsymbol{\theta}_i) + \beta p_i(\mathbf{x}) \quad (17.20)$$

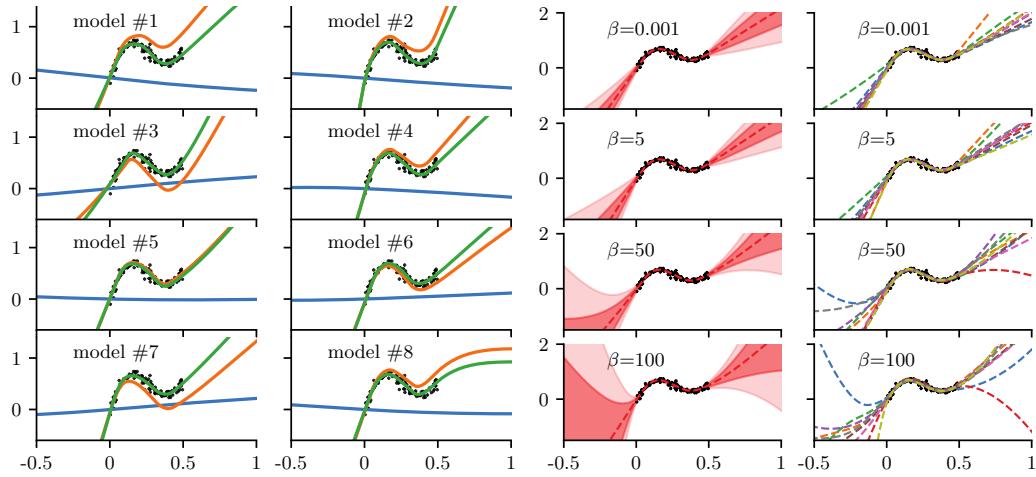


Figure 17.7: Deep ensemble with random priors. (a) Individual predictions from each member. Blue is the fixed random prior function, orange is the trainable function, green is the combination of the two. (b) Overall prediction from the ensemble, for increasingly large values of β . On the left we show (in red) the posterior mean and pointwise standard deviation, and on the right we show samples from the posterior. As β increases, we trust the random priors more, and pay less attention to the data, thus getting a more diffuse posterior. Generated by [randomized_priors.ipynb](#).

where $\beta \geq 0$ controls the amount of data-independent variation between the members. The trainable network learns to model the residual error between the true output and the value predicted by the prior. This is called a **random prior deep ensemble** [OAC18]. See Figure 17.7 for an illustration.

17.3.9.3 Deep ensembles as approximate Bayesian inference

The posterior predictive distribution for a Bayesian neural network cannot be expressed in closed form. Therefore all Bayesian inference approaches in deep learning are approximate. In this context, all approximate inference procedures fall onto a spectrum, representing how closely they approximate the true posterior predictive distribution. Deep ensembles can provide better approximations to a Bayesian model average than a single basin marginalization approach, because point masses from different basins of attraction represent greater functional diversity than standard Bayesian approaches which sample within a single basin.

17.3.9.4 Deep ensembles vs classical ensembles

Note that deep ensembles are slightly different from classical ensemble methods (see e.g., [Die00]), such as bagging and random forests, which obtain diversity of their predictors by training them on different subsets of the data (created using bootstrap resampling), or on different features. This data perturbation is necessary to get diversity when the base learner is a convex problem (such as a linear model, or shallow decision tree). In the deep ensemble approach, every model is trained on the same data, and the same input features. The diversity arises due to different starting parameters, different

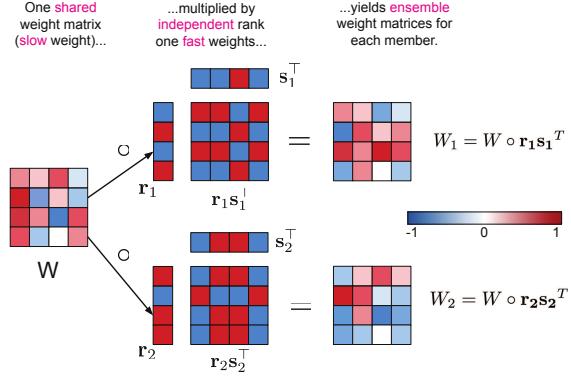


Figure 17.8: Illustration of batch ensemble with 2 ensemble members. From Figure 2 of [WTB20]. Used with kind permission of Paul Vicol.

random seeds, and SGD noise, which induces different solutions due to the nonconvex loss. It is also possible to explicitly enforce diversity of the ensemble members, which can provably improve performance [TB22].

17.3.9.5 Deep ensembles vs mixtures of experts and stacking

If we use weighted combinations of the models, $p(\boldsymbol{\theta}|\mathcal{D}) = \sum_{m=1}^M p(m|\mathcal{D})p(\boldsymbol{\theta}|m, \mathcal{D})$, where $p(m|\mathcal{D})$ is the marginal likelihood of model m , then, in the large sample limit, this mixture will concentrate on the MAP model, so only one component will be selected. By contrast, in deep ensembles, we always use M equally weighted models. Thus we see that Bayes model averaging is not the same as model ensembling [Min00b]. Indeed, ensembling can enlarge the expressive power of the posterior predictive distribution compared to BMA [OCM21].

We can also make the mixing weights be conditional on the inputs:

$$p(y|\mathbf{x}, \mathcal{D}) = \sum_m w_m(\mathbf{x}) p(y|\mathbf{x}, \boldsymbol{\theta}_m) \quad (17.21)$$

If we constrain the weights to be non-zero and sum to one, this is called a **mixture of experts**. However, if we allow a general positive weighted combination, the approach is called **stacking** [Wol92; Bre96; Yao+18a; CAII20]. In stacking, the weights $w_m(\mathbf{x})$ are usually estimated on hold-out data, to make the method more robust to model misspecification.

17.3.9.6 Batch ensemble

Deep ensembles require M times more memory and time than a single model. One way to reduce the memory cost is to share most of the parameters — which we call **slow weights**, \mathbf{W} — and then let each ensemble member m estimate its own local perturbation, which we will call **fast weights**, \mathbf{F}_m . We then define $\mathbf{W}_m = \mathbf{W} \odot \mathbf{F}_m$. For efficiency, we can define \mathbf{F}_m to be a rank-one matrix, $\mathbf{F}_m = \mathbf{s}_m \mathbf{r}_m^\top$, as illustrated in Figure 17.8. This is called **batch ensemble** [WTB20].

It is clear that the memory overhead is very small compared to naive ensembles, since we just need to store $2M$ vectors (\mathbf{s}_m^l and \mathbf{r}_m^l) for every layer l , which is negligible compared to the quadratic cost of storing the shared weight matrix \mathbf{W}^l .

In addition to memory savings, batch ensemble can reduce the inference time by a constant factor by leveraging within-device parallelism. To see this, consider the output of one layer using ensemble m on example n :

$$y_n^m = \varphi(\mathbf{W}_m^\top \mathbf{x}_n) = \varphi((\mathbf{W} \odot \mathbf{s}_m \mathbf{r}_m^\top)^\top \mathbf{x}_n) = \varphi((\mathbf{W}^\top (\mathbf{x}_n \odot \mathbf{s}_m) \odot \mathbf{r}_m)) \quad (17.22)$$

We can vectorize this for a minibatch of inputs \mathbf{X} by replicating \mathbf{r}_m and \mathbf{s}_m along the B rows in the batch to form matrices, giving

$$\mathbf{Y}_m = \varphi(((\mathbf{X} \odot \mathbf{S}_m) \mathbf{W}) \odot \mathbf{R}_m) \quad (17.23)$$

This applies the same ensemble parameters m to every example in the minibatch of size B . To achieve diversity during training, we can divide the minibatch into M sub-batches, and use sub-batch m to train \mathbf{W}_m . (Note that this reduces the batch size for training each ensemble to B/M .) At test time, when we want to average over M models, we can replicate each input M times, leading to a batch size of BM .

In [WTB20], they show that this method outperforms MC dropout at negligible extra memory cost. However, the best combination was to combine batch ensemble with MC dropout; in some cases, this approached the performance of naive ensembles.

17.3.10 Approximating the posterior predictive distribution

Once we have approximated the parameter posterior, $q(\boldsymbol{\theta}) \approx p(\boldsymbol{\theta}|\mathcal{D})$, we can use it to approximate the posterior predictive distribution:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int q(\boldsymbol{\theta}) p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) d\boldsymbol{\theta} \quad (17.24)$$

We often approximate this integral using Monte Carlo:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^s)) \quad (17.25)$$

where $\boldsymbol{\theta}^s \sim q(\boldsymbol{\theta}|\mathcal{D})$. We discuss some extensions of this approach below.

17.3.10.1 A linearized approximation

In [IKB21] they point out that samples from an approximate posterior, $q(\boldsymbol{\theta})$, can result in bad predictions when plugged into the model if the posterior puts probability density “in the wrong places”. This is because $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ is a highly nonlinear function of $\boldsymbol{\theta}$ that might behave quite differently when $\boldsymbol{\theta}$ is far from the MAP estimate on which $q(\boldsymbol{\theta})$ is centered. To avoid this problem, they propose to replace $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ with a linear approximation centered at the MAP estimate $\boldsymbol{\theta}^*$:

$$\mathbf{f}_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}^*) + \mathbf{J}(\mathbf{x})(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \quad (17.26)$$

where $\mathbf{J}_{\boldsymbol{\theta}^*}(\mathbf{x}) = \frac{\partial f(\mathbf{x}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}|_{\boldsymbol{\theta}^*}$ is the $P \times C$ Jacobian matrix, where P is the number of parameters, and C is the number of outputs. Such a model is well behaved around $\boldsymbol{\theta}^*$, and so the approximation

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \frac{1}{S} \sum_{s=1}^S p(\mathbf{y}|f_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}^s)) \quad (17.27)$$

often works better than Equation (17.25).

Note that $\mathbf{z} = f_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta})$ is a linear function of the parameters $\boldsymbol{\theta}$, but a nonlinear function of the inputs \mathbf{x} . Thus $p(\mathbf{y}|f_{\text{lin}}^{\boldsymbol{\theta}^*}(\mathbf{x}, \boldsymbol{\theta}))$ is a generalized linear model (Section 15.1), so [IKB21] call this approximation the **GLM predictive distribution**.

If we have a Gaussian approximation to the parameter, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then we can “push this through” the linear approximation to get

$$p(\mathbf{z}|\mathbf{x}, \mathcal{D}) \approx \mathcal{N}(\mathbf{z}|f(\mathbf{x}, \boldsymbol{\mu}), \mathbf{J}(\mathbf{x})^\top \boldsymbol{\Sigma} \mathbf{J}(\mathbf{x})) \quad (17.28)$$

where \mathbf{z} are the logits. (Alternatively, we can use the last layer method of Equation (17.17) to get a Gaussian approximation to $p(\mathbf{z}|\mathbf{x}, \mathcal{D})$.) If we approximate the final softmax layer with a probit function, we can analytically pass this Gaussian through the final softmax layer to deterministically compute the predictive probabilities $p(y=c|\mathbf{x}, \mathcal{D})$, using Equation (15.150). Alternatively, we can use the Laplace bridge approximation in Section 17.3.10.2.

17.3.10.2 The Laplace bridge approximation

Just using a point estimate of the probability of each class label, $p_c = p(y=c|\mathbf{x}, \mathcal{D})$, can be unreliable, since it does not convey any sense of uncertainty in the probability value, even though we may have taken the uncertainty of the parameters into account (e.g., using the methods of Section 17.3.10.1). An alternative is to represent the output over labels as a Dirichlet distribution, $\text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha})$, rather than a categorical distribution, $\text{Cat}(\mathbf{y}|\mathbf{p})$, where $\mathbf{p} = \text{softmax}(\mathbf{z})$. This is more appropriate if we view each datapoint as being annotated with a “soft” vector of probabilities (e.g., representing consensus votes from human raters), rather than a one-hot encoding with a single “ground truth” value. This can be useful for settings where the true label is ambiguous (see e.g., [Bey+20; Dum+18]).

We can either train the model to predict the Dirichlet parameters directly (as in the **prior network** approach of [MG18]), or we can train the model to predict softmax outputs in the usual way, and then derive the Dirichlet parameters from a Gaussian approximation to the posterior. The latter approach is known as the **Laplace bridge** [HKh22], and has the advantage that it can be used as a post-processing method. It works as follows. First we compute a Gaussian approximation to the logits, $p(\mathbf{z}|\mathbf{x}, \mathcal{D}) = \mathcal{N}(\mathbf{z}|\mathbf{m}, \mathbf{V})$ using Equation (17.28) or Equation (17.17). Then we compute

$$\alpha_i = \frac{1}{V_{ii}} \left(1 - \frac{2}{C} + \frac{\exp(m_i)}{C^2} \sum_{j=1}^C \exp(-m_j) \right) \quad (17.29)$$

where C is the number of classes. We can then derive the probabilities of each class label using $p_c = \mathbb{E}[\pi_c] = \alpha_c/\alpha_0$, where $\alpha_0 = \sum_{c=1}^C \alpha_c$.

Note that the derivation of the above result assumes that the Gaussian terms sum to zero, since the Gaussian has one less degree of freedom compared to the Dirichlet. To ensure this, it is necessary

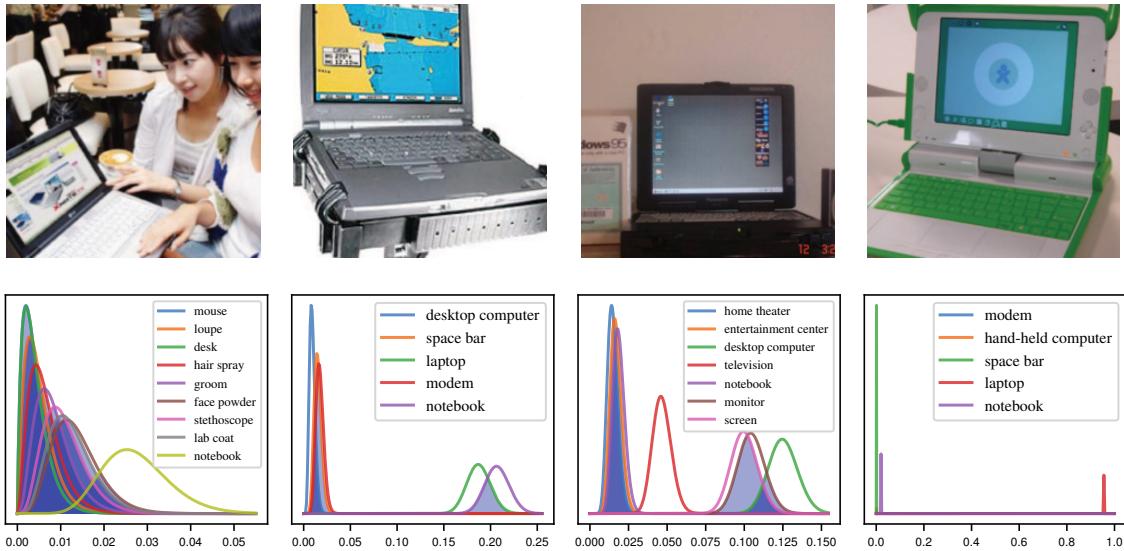


Figure 17.9: Illustration of uncertainty about individual labels in an image classification problem. Top row: images from the “laptop” class of ImageNet. Bottom row: beta marginals for the top- k predictions for the respective image. First column: high uncertainty about all the labels. Second column: “notebook” and “laptop” have high confidence. Third column: “desktop”, “screen” and “monitor” have high confidence. Fourth column: only “laptop” has high confidence. (Compare to Figure 14.4.) From Figure 6 of [HKh22]. Used with kind permission of Philipp Hennig.

to first project the Gaussian distribution onto this constraint surface, yielding

$$p(\mathbf{z}|\mathbf{x}, \mathcal{D}) = \mathcal{N}\left(\mathbf{z}|\mathbf{m} - \frac{\mathbf{V}\mathbf{1}\mathbf{1}^\top \mathbf{m}}{\mathbf{1}^\top \mathbf{V}_* \mathbf{1}}, \mathbf{V} - \frac{\mathbf{V}\mathbf{1}\mathbf{1}^\top \mathbf{V}}{\mathbf{1}^\top \mathbf{V}\mathbf{1}}\right) = \mathcal{N}(\mathbf{z}|\mathbf{m}', \mathbf{V}') \quad (17.30)$$

where $\mathbf{1}$ is the ones vector of size C . To avoid potential problems where $\boldsymbol{\alpha}$ is sparse, [HKh22] propose to also scale the posterior (after the zero-sum constraint) by using $\mathbf{m}'' = \mathbf{m}'/\sqrt{c}$ and $\mathbf{V}'' = \mathbf{V}'/c$, where $c = (\sum_{ii} V'_{ii})/\sqrt{C/2}$.

One useful property of the Laplace bridge approximation, compared to the probit approximation, is that we can easily compute a marginal distribution over the probability of each label being present. This is because the marginals of a Dirichlet are beta distributions. We can use this to adaptively compute a top- k prediction set; this is similar in spirit to conformal prediction (Section 14.3.1), but is Bayesian, in the sense that it represents per-instance uncertainty. The method works as follows. First we sort the class labels in decreasing order of expected probability, to get $\tilde{\boldsymbol{\alpha}}$; next we compute the marginal distribution over the probability for the top label,

$$p(\pi_1|\mathbf{x}, \mathcal{D}) = \text{Beta}(\tilde{\alpha}_1, \alpha_0 - \tilde{\alpha}_1) \quad (17.31)$$

where $\alpha_0 = \sum_c \alpha_c$. We then compute the marginal distributions for the other labels in a similar way,

and return all labels that have significant overlap with the top label. As we see from the examples in Figure 17.9, this approach can return variable-sized outputs, reflecting uncertainty in a natural way.

17.3.10.3 Distillation

The MC approximation to the posterior predictive is S times slower than a standard, deterministic plug-in approximation. One way to speed this up is to use **distillation** to approximate the semi-parametric “teacher” model p_t from Equation (17.25) by a parametric “student” model p_s by minimizing $\mathbb{E}[D_{\text{KL}}(p_t(\mathbf{y}|\mathbf{x}) \parallel p_s(\mathbf{y}|\mathbf{x}))]$ wrt p_s . This approach was first proposed in [HVD14], who called the technique “**dark knowledge**”, because the teacher has “hidden” information in its predictive probabilities (logits) than is not apparent in the raw one-hot labels.

In [Kor+15], this idea was used to distill the predictions from a teacher whose parameter posterior was computed using HMC; this is called “**Bayesian dark knowledge**”. A similar idea was used in [BPK16; GBP18], who distilled the predictive distribution derived from MC dropout (Section 17.3.1).

Since the parametric student is typically less flexible than the semi-parametric teacher, it may be overconfident, and lack diversity in its predictions. To avoid this overconfidence, it is safer to make the student be a mixture distribution [SG05; Tra+20a].

17.3.11 Tempered and cold posteriors

When working with BNNs for classification problems, the likelihood is usually taken to be

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\text{softmax}(f(\mathbf{x}; \boldsymbol{\theta}))) \quad (17.32)$$

where $f(\mathbf{x}; \boldsymbol{\theta}) \in \mathbb{R}^C$ returns the logits over the C class labels. This is the same as in multinomial logistic regression (Section 15.3.2); the only difference is that f is a nonlinear function of $\boldsymbol{\theta}$.

However, in practice, it is often found (see e.g., [Zha+18; Wen+20b; LST21; Noc+21]) that BNNs give better predictive accuracy if the likelihood function is scaled by some power α . That is, instead of targeting the posterior $p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta})$, these methods target the **tempered posterior**, $p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})^\alpha p(\boldsymbol{\theta})$. In log space, we have

$$\log p_{\text{tempered}}(\boldsymbol{\theta}|\mathcal{D}) = \alpha \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const} \quad (17.33)$$

This is also called an **α -posterior** or **power posterior** [Med+21].

Another common method is to target the **cold posterior**, $p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) \propto p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})^{1/T}$, or, in log space,

$$\log p_{\text{cold}}(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{T} \log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) + \frac{1}{T} \log p(\boldsymbol{\theta}) + \text{const} \quad (17.34)$$

If $T < 1$, we say that the posterior is “cold”. Note that, in the case of a Gaussian prior, using the cold posterior is the same as using the tempered posterior with a different hyperparameter, since $\frac{1}{T} \log p_{\text{cold}}(\boldsymbol{\theta})$ is given by

$$\frac{1}{T} \log \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{cold}}^2 \mathbf{I}) = -\frac{1}{2T\sigma_{\text{cold}}^2} \sum_i \theta_i^2 + \text{const} = \mathcal{N}(\boldsymbol{\theta}|0, \sigma_{\text{tempered}}^2 \mathbf{I}) + \text{const} \quad (17.35)$$

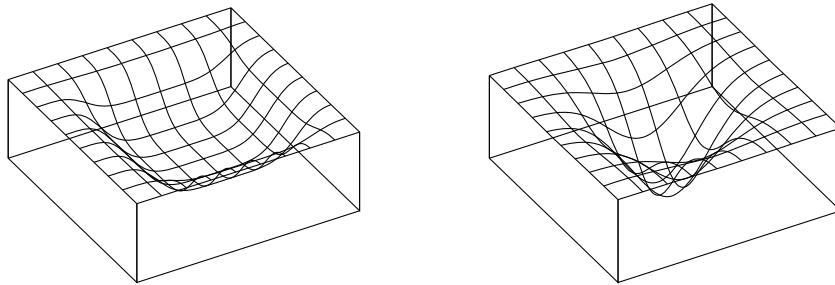


Figure 17.10: Flat vs sharp minima. From Figures 1 and 2 of [HS97]. Used with kind permission of Jürgen Schmidhuber.

which equals $\log p_{\text{tempered}}(\theta)$ if we set $\sigma_{\text{tempered}}^2 = T\sigma_{\text{cold}}^2$. Thus both methods are effectively the same, and just reweight the likelihood by $\alpha = 1/T$.

Cold posteriors in Bayesian neural network classifiers are a consequence of underrepresenting aleatoric (label) uncertainty, as shown by [Kap+22]. On benchmarks such as CIFAR-100, we should have essentially no uncertainty about the labels of the training images, yet Bayesian classifiers with softmax likelihoods have very high uncertainty for these points. Moreover, [Izm+21b] showed that the cold posterior effect in all the examples of [Wen+20b] when data augmentation is removed. [Kap+22] show that with the SGLD inference in [Wen+20b], data augmentation has the effect of raising the likelihood to a power $1/K$ for minibatches of size K . Cold posteriors exactly counteract this effect, more honestly representing our beliefs about aleatoric uncertainty, by sharpening the likelihood. However, tempering is not required, and [Kap+22] show that by using a Dirichlet observation model to explicitly represent (lack of) label noise, there is no cold posterior effect, even with data augmentation. The curation hypotheses of [Ait21] can be considered a special case of the above explanation, where curation has the effect of increasing our confidence about training labels.

In Section 14.1.3, we discuss generalized variational inference, which gives a general framework for understanding whether and how the likelihood or prior could benefit from tempering. Tempering is particularly useful if (as is usually the case) the model is misspecified [KJD21].

17.4 Generalization in Bayesian deep learning

In this section, we discuss why “being Bayesian” can improve predictive accuracy and generalization performance.

17.4.1 Sharp vs flat minima

Some optimization methods (in particular, second-order batch methods) are able to find “needles in haystacks”, corresponding to narrow but deep “holes” in the loss landscape, corresponding to parameter settings with very low loss. These are known as **sharp minima**, see Figure 17.10(right). From the point of view of minimizing the empirical loss, the optimizer has done a good job. However, such solutions generally correspond to a model that has overfit the data. It is better to find points that correspond to **flat minima**, as shown in Figure 17.10(left); such solutions are more robust and

generalize better. To see why, note that flat minima correspond to regions in parameter space where there is a lot of posterior uncertainty, and hence samples from this region are less able to precisely memorize irrelevant details about the training set [AS17]. Put another way, the description length for sharp minima is large, meaning you need to use many bits of precision to specify the exact location in parameter space to avoid incurring large loss, whereas the description length for flat minima is less, resulting in better generalization [Mac03].

SGD often finds such flat minima by virtue of the addition of noise, which prevents it from “entering” narrow regions of the loss landscape (see Section 12.5.7). In addition, in higher dimensional spaces, flat regions occupy a much greater volume, and are thus much more easily discoverable by optimization procedures. More precisely, the analysis in [SL18] shows that the probability of entering any given basin of attraction \mathcal{A} around a minimum is given by $p_{SGD}(\boldsymbol{\theta} \in \mathcal{A}) \propto \int_{\mathcal{A}} e^{-\mathcal{L}(\boldsymbol{\theta})} d\boldsymbol{\theta}$. Note that this is integrating over the volume of space corresponding to \mathcal{A} , and hence is proportional to the model evidence (marginal likelihood) for that region, as explained in Section 3.8.1. Since the evidence is parameterization invariant (since we marginalize out the parameters), this means that SGD will avoid regions that have low evidence (corresponding to sharp minima) regardless of how we parameterize the model (contrary to the claims in [Din+17]).

In fact, several papers have shown that we can view SGD as approximately sampling from the Bayesian posterior (see Section 17.3.8). The SWA method (Section 17.3.8) can be seen as finding a center of mass in the posterior based on these SGD samples, finding solutions that generalize better than picking a single SGD point.

If we must use a single solution, a flat one will help us better approximate the Bayesian model average in the integral of Equation (17.1). However, by attempting to perform a more complete Bayesian model average, we will select for flatness without having to deal with the messiness of having to worry about flatness definitions, or the effects of reparameterization, or unknown implicit regularization, as the model average will automatically weight regions with the greatest volume.

17.4.2 Mode connectivity and the loss landscape

In DNNs there are often many low-loss solutions, which provide complementary explanations of the data. Moreover, in [Gar+18c] they showed that two independently trained SGD solutions can be connected by a curve in a subspace, along which the training loss remains near-zero, known as **mode connectivity**. Despite having the same training loss, these different parameter settings give rise to very different functions, as illustrated in Figure 17.11, where we show predictions on a 1d regression problem coming from different points in parameter space obtained by interpolating along a mode connecting curve between two distinct MAP estimates. Using a Bayesian model average, we can combine these functions together to provide much better performance over a single flat solution [Izm+19].

Recently, it has been discovered [Ben+21b] that there are in fact large multidimensional simplexes of low loss solutions, which can be combined together for significantly improved performance. These results further motivate the Bayesian approach (Equation (17.1)), where we perform a posterior weighted model average.

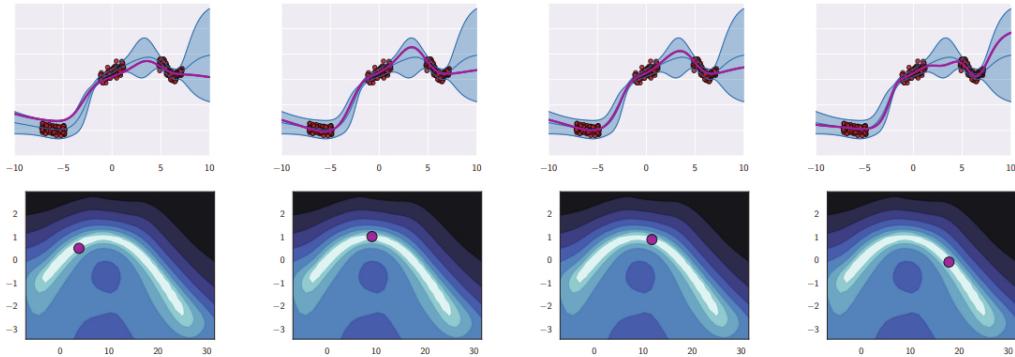


Figure 17.11: Diversity of high performing functions sampled from the posterior. Top row: we show predictions on the 1d input domain for 4 different functions. We see that they extrapolate in different ways outside of the support of the data. Bottom row: we show a 2d subspace spanning two distinct modes (MAP estimates), and connected by a low-loss curved path computed as in [Gar+18c]. From Figure 8 of [WI20]. Used with kind permission of Andrew Wilson.

17.4.3 Effective dimensionality of a model

Modern DNNs have millions of parameters, but these parameters are often not well-determined by the data, i.e., there can be a lot of posterior uncertainty. By averaging over the posterior, we reduce the chance of overfitting, because we do not use “degrees of freedom” that are not needed or warranted.

To quantify the number of degrees of freedom, or **effective dimensionality** [Mac92b], we follow [MBW20] and define

$$N_{\text{eff}}(\mathbf{H}, c) = \sum_{i=1}^k \frac{\lambda_i}{\lambda_i + c}, \quad (17.36)$$

where λ_i are the eigenvalues of the Hessian matrix \mathbf{H} computed at a local mode, and $c > 0$ is a regularization parameter. Intuitively, the effective dimension counts the number of well-determined parameters. A “flat minimum” will have many directions in parameter space that are not well-determined, and hence will have low effective dimensionality. This means that we can perform Bayesian inference in a low dimensional subspace [Izm+19]: Since there is functional homogeneity in all directions but those defining the effective dimension, neural networks can be significantly compressed.

This compression perspective can also be used to understand why the effective dimension can be a good proxy for generalization. If two models have similar training loss, but one has lower effective dimension, then it is providing a better compression for the data at the same fidelity. In Figure 17.12 we show that for CNNs with low training loss (above the green partition), the effective dimensionality closely tracks generalization performance. We also see that the number of parameters alone is not a strong determinant of generalization. Indeed, models with more parameters can have a lower number of effective parameters. We also see that wide but shallow models overfit, while depth helps provide

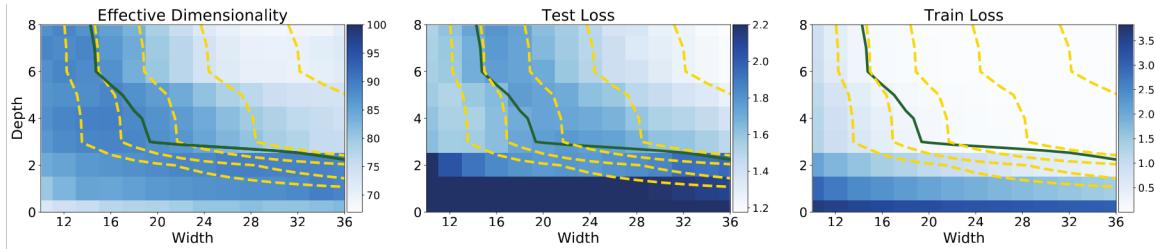


Figure 17.12: Left: effective dimensionality as a function of model width and depth for a CNN on CIFAR-100. Center: test loss as a function of model width and depth. Right: train loss as a function of model width and depth. Yellow level curves represent equal parameter counts ($1e5$, $2e5$, $4e5$, $1.6e6$). The green curve separates models with near-zero training loss. Effective dimensionality serves as a good proxy for generalization for models with low train loss. We see wide but shallow models overfit, providing low train loss, but high test loss and high effective dimensionality. For models with the same train loss, lower effective dimensionality can be viewed as a better compression of the data at the same fidelity. Thus depth provides a mechanism for compression, which leads to better generalization. From Figure 2 of [MBW20]. Used with kind permission of Andrew Wilson.

lower effective dimensionality, leading to a better compression of the data. It is depth that makes modern neural networks distinctive, providing hierarchical inductive biases making it possible to discover more regularity in the data.

17.4.4 The hypothesis space of DNNs

Zhang et al. [Zha+17] showed that CNNs can fit CIFAR-10 images with random labels with zero training error, but can still generalize well on the noise-free test set. It has been claimed that this result contradicts a classical understanding of generalization, because it shows that neural networks are capable of significantly overfitting the data, but can still generalize well on structured inputs.

We can resolve this paradox by taking a Bayesian perspective. In particular, we know that modern CNNs are very flexible, so they can fit almost pattern (since they are in fact universal approximators). However, their architecture encodes a prior over what kinds of patterns they expect to see in the data (see Section 17.2.5). Image datasets with random labels *can* be represented by this function class, but such solutions receive very low marginal likelihood, since they strongly violate the prior assumptions [WI20]. By contrast, image datasets where the output labels are consistent with patterns in the input get much higher marginal likelihood.

This phenomenon is not unique to DNNs. For example, it also occurs with Gaussian processes (Chapter 18). Such models are also universal approximators, but they allocate most of their probability mass to a small range of solutions (depending on the chosen kernel). They can also fit image datasets with random labels, but such data receives a low marginal likelihood [WI20].

In general, we can distinguish the support of a model, i.e., the set of functions it can represent, from the distribution over that support, i.e., the inductive bias which leads it to prefer some functions over others. We would like to use models where the support is large, so we can capture the complexity of real-world data, but also where the inductive bias places probability mass on the kinds of functions we expect to see. If we succeed at this, the posterior will quickly converge on the true function after

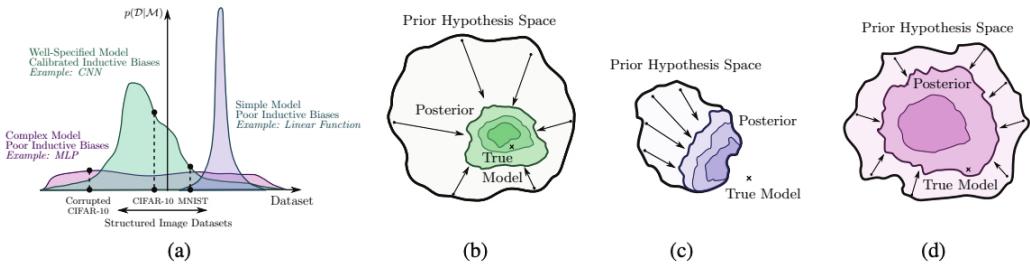


Figure 17.13: Illustration of the behavior of different kinds of model families and the prior distributions they induce over datasets. (a) The purple model is a simple linear model that has small support, and can only represent a few kinds of datasets. The pink model is an unstructured MLP: this has support over a large range of datasets with a fairly uninformative (broad) prior. Finally the green model is a CNN; this has support over a large range of datasets but the prior is more concentrated on certain kinds of datasets that have compositional structure. (b) The posterior for the green model (CNN) rapidly collapses to the true model, since it is consistent with the data. (c) The posterior for the purple model (linear) also rapidly collapses, but to a solution which cannot represent the true model. (d) The posterior for the pink model (MLP) collapses very slowly (as a function of dataset size). From Figure 2 of [WI20]. Used with kind permission of Andrew Wilson.

seeing a small amount of data. This idea is sketched in Figure 17.13.

17.4.5 PAC-Bayes

PAC-Bayes [McA99; LC02; Gue19; Alq21; GSZ21] provides a promising mechanism to derive non-vacuous generalization bounds for large *stochastic networks* [Ney+17; NBS18; DR17], with parameters sampled from a probability distribution. In particular, the difference between the train error and the generalization error can be expressed as

$$\sqrt{\frac{D_{\text{KL}}(Q \parallel P) + c}{2(N-1)}}, \quad (17.37)$$

where c is a constant, N is the number of training points, P is the prior distribution over the parameters, and Q is an arbitrary distribution, which can be chosen to optimize the bound.

The perspective in this chapter is largely complementary, and in some ways orthogonal, to the PAC-Bayes literature. Our focus has been on Bayesian marginalization, particularly multi-modal marginalization, and a prescriptive approach to model construction. In contrast, PAC-Bayes bounds are about bounding the empirical risk of a single sample, rather than marginalization, and are not currently prescriptive: what we would do to improve the bounds, such as reducing the number of model parameters, or using highly compact priors, does not typically improve generalization. Moreover, while we have seen Bayesian model averaging over multimodal posteriors has a significant effect on generalization, it has a minimal logarithmic effect on PAC-Bayes bounds. In general, because the bounds are loose, albeit non-vacuous in some cases, there is often room to make modeling choices that improve PAC-Bayes bounds without improving generalization, making it hard to derive a prescription for model construction from the bounds.

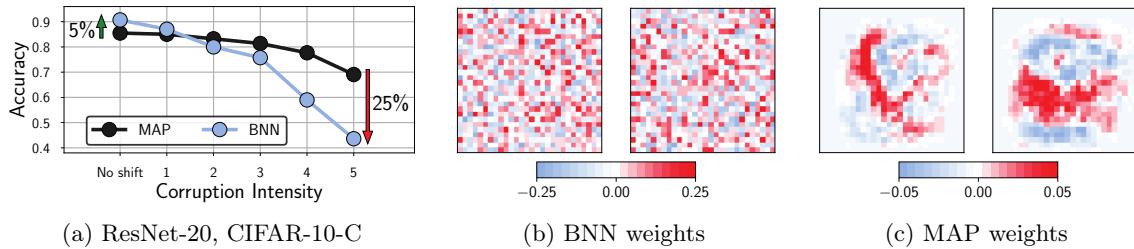


Figure 17.14: Bayesian neural networks under covariate shift. a: Performance of a ResNet-20 on the pixelate corruption in CIFAR-10-C. For the highest degree of corruption, a Bayesian model average underperforms a MAP solution by 25% (44% against 69%) accuracy. See Izmailov et al. [Izm+21b] for details. b: Visualization of the weights in the first layer of a Bayesian fully-connected network on MNIST sampled via HMC. c: The corresponding MAP weights. We visualize the weights connecting the input pixels to a neuron in the hidden layer as a 28×28 image, where each weight is shown in the location of the input pixel it interacts with. This is Figure 1 of Izmailov et al. [Izm+21a].

17.4.6 Out-of-distribution generalization for BNNs

Bayesian methods are often assumed to be more robust in the context of distribution shift (discussed in Chapter 19), because they capture more uncertainty than methods based on point estimation. However, there are some subtleties, some of which we discuss below.

17.4.6.1 BMA can give poor results with default priors

Many approximate inference methods, especially deep ensembles, are significantly less overconfident (more well calibrated) in the presence of some kinds of covariate shifts [Ova+19]. However, in [Izm+21b], it was noted that HMC, which arguably offers the most accurate approximation to the posterior, often works poorly under distribution shift.

Rather than an idiosyncracy of HMC, Izmailov et al. [Izm+21a] show this lack of robustness is a foundational issue of Bayesian model averaging under covariate shift, caused by degeneracies in the training data, and a poor choice of prior. As an illustrative special case, MNIST digits all have black corner pixels. Weights in the first layer of a neural network connected to these pixels are multiplied by zero, and thus can take any value without affecting the outputs of the network. Classical MAP training or deep ensembles of MAP solutions with a Gaussian prior will therefore drive these parameters to zero, since they don't help with the data fit, and the resulting network will be robust to corruptions on these pixels. On the other hand, the posterior for these parameters will be the same as the prior, and so a Bayesian model average will multiply corruptions by random numbers sampled from the prior, leading to degraded predictive performance.

Figure 17.14(b, c) visualizes this example, showing the first-layer weights of a fully-connected network for the MAP solution and a BNN posterior sample, on MNIST. The MAP weights corresponding to zero intensity pixels near the boundary are near zero, while the BNN weights look noisy, sampled from a Gaussian prior.

Izmailov et al. [Izm+21a] prove that this issue is a special case of a much more general problem, whenever there are linear dependencies in the input features of the training data, both for fully-

connected and convolutional networks. In this case, the data live on a hyperplane. If a covariate or domain shift, moves orthogonal to this hyperplane, the posterior will be the same as the prior in the direction of the shift. The posterior model average will thus be highly vulnerable to shifts that do not particularly affect the underlying semantic structure of the problem (such as corruptions), whereas the MAP solution will be entirely robust to such shifts.

By introducing a prior over parameters which is aligned with the principal components of the training inputs, we can substantially improve the generalization accuracy of Bayesian neural networks in out-of-distribution settings. Izmailov et al. [Izm+21a] propose the following *EmpCov* prior: $p(w^1) = \mathcal{N}(0, \alpha\Sigma + \epsilon I)$, where w^1 are the first layer weights, $\Sigma = \frac{1}{n-1} \sum_{i=1}^n x_i x_i^T$ is the empirical covariance of the training input features x_i , $\alpha > 0$ determines the scale of the prior, and ϵ is a small positive constant to ensure the covariance matrix is positive definite. With this improved prior they are able to obtain a method that is much more robust to distribution shift.

17.4.6.2 BNNs can be overconfident on OOD inputs

An important problem in practice is how a predictive model will behave when it is given an input that is “out of distribution” or OOD. Ideally we would like the model to express that it is not confident in its prediction, so that the system can abstain from predicting (see Section 19.3.3). Using “exact” inference methods, such as MCMC, for BNNs can give this behavior in some cases. For example, in Section 19.3.3.1 we showed that an MLP which was fit to MNIST using SGLD would be less overconfident than a point estimate (computed using SGD) when presented with inputs from fashion MNIST. However, this behavior does not always occur reliably.

To illustrate the problem, consider the 2d nonlinear binary classification dataset shown in Figure 17.15. In addition to the two training classes, we have highlighted (in green) a set of OOD inputs that are far from the support of the training set. Intuitively we would expect the model to predict a probability of 0.5 (corresponding to “don’t know”) for such inputs that are far from the training set. However we see that the only methods that do so are the Gaussian process (GP) classifier (see Section 18.4) and the SNGP model (Section 17.3.6), which contains a GP layer on top of the feature extractor.

The lesson we learn from this simple example is that “being Bayesian” only helps if we are using a good hypothesis class. If we only consider a single MLP classifier, with standard Gaussian priors on the weights, it is extremely unlikely that we will learn the kind of compact decision boundary shown in Figure 17.15g, because that function has negligible support under our prior (c.f. Section 17.4.4). Instead we should embrace the power of Bayes to avoid overfitting and use as complex a model class as we can afford.

17.4.7 Model selection for BNNs

Historically, the marginal likelihood (aka Bayesian evidence) has been used for model selection problems, such as choosing neural architectures or hyperparameter values [Mac92a]. Recent methods based on the Laplace approximation, such as [Imm+21; Dax+21], have made this scalable to large BNNs. However, [Lot+22] argue that it is much better to use the conditional marginal likelihood, which we discuss in Section 3.8.5.

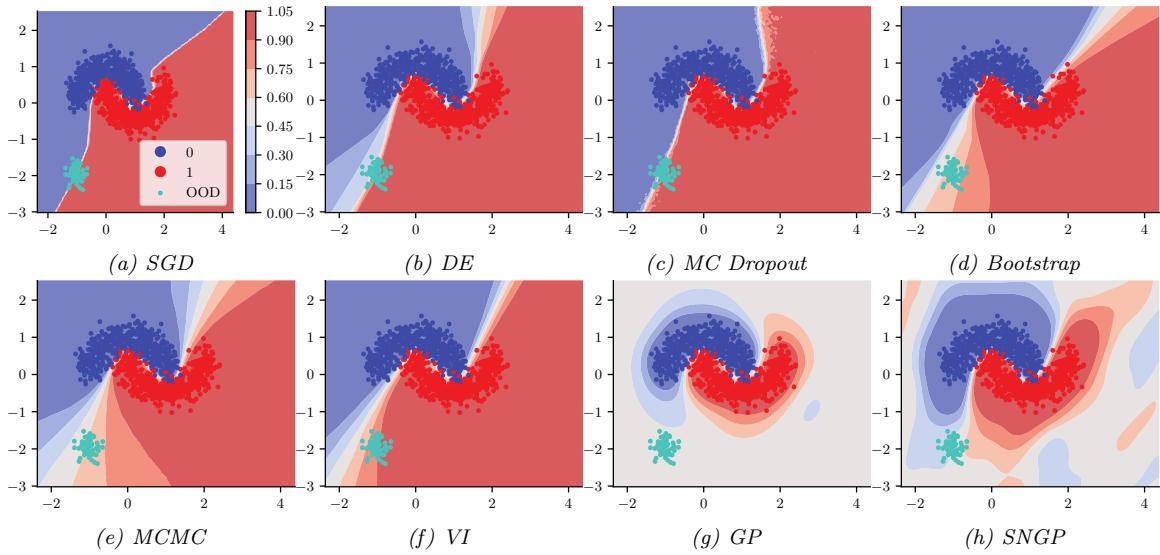


Figure 17.15: Predictions made by various (B)NNs when presented with the training data shown in blue and red. The green blob is an example of some OOD inputs. Methods are: (a) standard SGD; (b) deep Ensemble of 10 models with different random initializations; (c) MC dropout with 50 samples; (d) bootstrap training, where each of the 10 models is initialized identically but given different versions of the data, obtained by resampling with replacement; (e) MCMC using NUTS algorithm with 3000 warmup steps and 3000 samples; (f) variational inference; (g) Gaussian process classifier using RBF kernel; (h) SNGP. The model is an MLP with 8,16,16,8 units in the hidden layers and ReLu activation. The output layer has 1 neuron with sigmoid activation. Generated by [makemoons_comparison.ipynb](#)

17.5 Online inference

In Section 17.3, we have focused on batch or offline inference. However, an important application of Bayesian inference is in sequential settings, where the data arrives in a continuous stream, and the model has to “keep up”. This is called **sequential Bayesian inference**, and is one approach to **online learning** (see Section 19.7.5). In this section, we discuss some algorithmic approaches to this problem in the context of DNNs. These methods are widely used for continual learning, which we discuss Section 19.7.

17.5.1 Sequential Laplace for DNNs

In [RBB18b], they extended the Laplace method of Section 17.3.2 to the sequential setting. Specifically, let $p(\boldsymbol{\theta}|\mathcal{D}_{1:t-1}) \approx \mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}_{t-1}, \boldsymbol{\Lambda}_{t-1}^{-1})$ be the approximate posterior from the previous step; we assume the precision matrix is Kronecker factored. We now compute the new mean by solving the MAP

problem

$$\boldsymbol{\mu}_t = \operatorname{argmax} \log p(\mathcal{D}_t | \boldsymbol{\theta}) + \log p(\boldsymbol{\theta} | \mathcal{D}_{t-1}) \quad (17.38)$$

$$= \operatorname{argmax} \log p(\mathcal{D}_t | \boldsymbol{\theta}) - \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \boldsymbol{\Lambda}_{t-1}^{-1} (\boldsymbol{\theta} - \boldsymbol{\mu}_{t-1}) \quad (17.39)$$

Once we have computed $\boldsymbol{\mu}_t$, we compute the approximate Hessian at this point, and get the new posterior precision

$$\boldsymbol{\Lambda}_t = \lambda \mathbf{H}(\boldsymbol{\mu}_t) + \boldsymbol{\Lambda}_{t-1} \quad (17.40)$$

where $\lambda \geq 0$ is a weighting factor that trades off how much the model pays attention to the new data vs old data.

Now suppose we use a diagonal approximation to the posterior prediction matrix. From Equation (17.39), we see that this amounts to adding a quadratic penalty to each new MAP estimate, to encourage it to remain close to the parameters from previous tasks. This approach is called **elastic weight consolidation (EWC)** [Kir+17].

17.5.2 Extended Kalman filtering for DNNs

In Section 29.7.2, we showed how Kalman filtering can be used to incrementally compute the exact posterior for the weights of a linear regression model with known variance, i.e., we compute $p(\boldsymbol{\theta} | \mathcal{D}_{1:t}, \sigma^2)$, where $\mathcal{D}_{1:t} = \{(\mathbf{u}_i, y_i) : i = 1 : t\}$ is the data seen so far, and

$$p(y_t | \mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t | \boldsymbol{\theta}^\top \mathbf{u}_t, \sigma^2) \quad (17.41)$$

is the linear regression likelihood. The application of KF to this model is known as recursive least squares.

Now consider the case of nonlinear regression:

$$p(y_t | \mathbf{u}_t, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(y_t | f(\boldsymbol{\theta}, \mathbf{u}_t), \sigma^2) \quad (17.42)$$

where $f(\boldsymbol{\theta}, \mathbf{u}_t)$ is some nonlinear function, such as an MLP. We can use the extended Kalman filter (Section 8.3.2) to approximately compute $p(\boldsymbol{\theta}_t | \mathcal{D}_{1:t}, \sigma^2)$, where $\boldsymbol{\theta}_t$ is the hidden state (see e.g., [SW89; PF03]). To see this, note that we can set the dynamics model to the identity function, $f(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t$, so the parameters are propagated through unchanged, and the observation model to the input-dependent function $f(\boldsymbol{\theta}_t) = f(\boldsymbol{\theta}_t, \mathbf{u}_t)$. We set the observation noise to $\mathbf{R}_t = \sigma^2$, and the dynamics noise to $\mathbf{Q}_t = q\mathbf{I}$, where q is a small constant, to allow the parameters to slowly drift according to artificial **process noise**. (In practice it can be useful to anneal q from a large initial value to something near 0.)

17.5.2.1 Example

We now give an example of this process in action. We sample a synthetic dataset from the true function

$$h^*(u) = x - 10 \cos(u) \sin(u) + u^3 \quad (17.43)$$

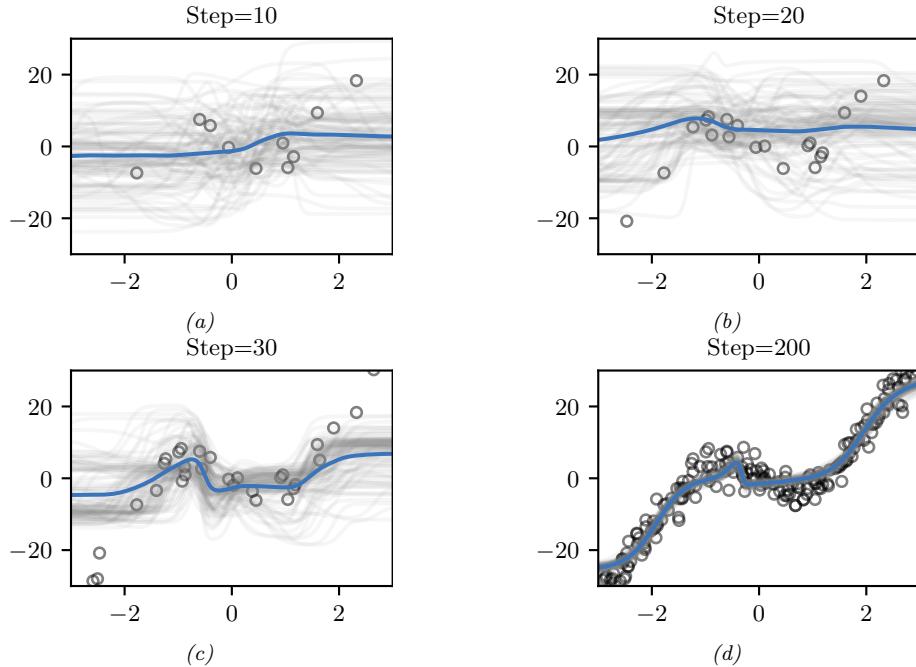


Figure 17.16: Sequential Bayesian inference for the parameters of an MLP using the extended Kalman filter. We show results after seeing the first 10, 20, 30 and 200 observations. (For a video of this, see <https://bit.ly/3wXnWaM>.) Generated by `ekf_mlp.ipynb`.

and add Gaussian noise with $\sigma = 3$. We then fit this with an MLP with one hidden layer with H hidden units, so the model has the form

$$f(\boldsymbol{\theta}, \mathbf{u}) = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{u} + \mathbf{b}_1) + \mathbf{b}_2 \quad (17.44)$$

where $\mathbf{W}_1 \in \mathbb{R}^{H \times 1}$, $\mathbf{b}_1 \in \mathbb{R}^H$, $\mathbf{W}_2 \in \mathbb{R}^{1 \times H}$, $\mathbf{b}_2 \in \mathbb{R}^1$. We set $H = 6$, so there are $D = 19$ parameters in total.

Given the data, we sequentially compute the posterior, starting from a vague Gaussian prior, $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta} | \mathbf{0}, \boldsymbol{\Sigma}_0)$, where $\boldsymbol{\Sigma}_0 = 100\mathbf{I}$. (In practice we cannot start from the prior mean, which is $\boldsymbol{\theta}_0 = \mathbf{0}$, since linearizing the model around this point results in a zero gradient, so we use an initial random sample for $\boldsymbol{\theta}_0$.) The results are shown in Figure 17.16. We can see that the model adapts to the data, without having to specify any learning rate. In addition, we see that the predictions become gradually more confident, as the posterior concentrates on the MLE.

17.5.2.2 Setting the variance terms

In the above example, we set the variance terms by hand. In general we need to estimate the noise variance σ , which determines \mathbf{R}_t and hence the learning rate, as well as the strength of the prior $\boldsymbol{\Sigma}_0$, which controls the amount of regularization. Some methods for doing this are discussed in [FNG00].

17.5.2.3 Reducing the computational complexity

The naive EKF method described above takes $O(N_z^3)$ time, which is prohibitive for large neural networks. A simple approximation, known as the **decoupled EKF**, was proposed in [PF91; SPD92] (see [PF03] for a review). This partitions the weights into G groups or blocks, and estimates the relevant matrices for each group g independently. If $G = 1$, this reduces the standard global EKF. If we put each weight into its own group, we get a fully diagonal approximation. In practice this does not work any better than SGD, since it ignores correlations between the parameters. A useful compromise is to put all the weights corresponding to each neuron into its own group; this is called **node decoupled EKF**, which has been used in [Sim02] to train RBF networks and [GUK21] to train exponential family matrix factorization models (widely used in recommender systems). For more details on DEKF, [Supplementary](#) Section 17.1.

Another approach to increasing computational efficiency is to leverage the fact that the effective dimensionality of a DNN is often quite low (see Section 17.4.3). Indeed we can approximate the model parameters by using a low dimensional vector of coefficients that specify the point in a linear manifold corresponding to weight space; the basis set defining this linear manifold can either be chosen randomly [Li+18b; GARD18; Lar+22], or can be estimated using PCA applied to the SGD iterates [Izm+19]. We can exploit this observation to perform EKF in this low-dimensional subspace, which significantly speeds up inference, as discussed in [DMKM22].

17.5.3 Assumed density filtering for DNNs

In Section 8.6.3, we discussed how to use assumed density filtering (ADF) to perform online (binary) logistic regression. In this section, we generalize this to nonlinear predictive models, such as DNNs. The key is to perform Gaussian moment matching of the hidden activations at each layer of the model. This provides an alternative to the EKF approach in Section 17.5.2, which is based on linearization of the network.

We will assume the following likelihood:

$$p(\mathbf{y}_t | \mathbf{u}_t, \mathbf{w}_t) = \text{Expfam}(\mathbf{y}_t | \ell^{-1}(f(\mathbf{u}_t; \mathbf{w}_t))) \quad (17.45)$$

where $f(\mathbf{x}; \mathbf{w})$ is the DNN, ℓ^{-1} is the inverse link function, and $\text{Expfam}()$ is some exponential family distribution. For example, if f is linear and we are solving a binary classification problem, we can write

$$p(y_t | \mathbf{u}_t, \mathbf{w}_t) = \text{Ber}(y_t | \sigma(\mathbf{u}_t^\top \mathbf{w}_t)) \quad (17.46)$$

We discussed using ADF to fit this model in Section 8.6.3.

In [HLA15b], they propose **probabilistic backpropagation (PBP)**, which is an instance of ADF applied to MLPs. The basic idea is to approximate the posterior over the weights in each layer using a fully factorized distribution

$$p(\mathbf{w}_t | \mathcal{D}_{1:t}) \approx p_t(\mathbf{w}_t) = \prod_{l=1}^L \prod_{i=1}^{D_l} \prod_{j=1}^{D_{l-1}+1} \mathcal{N}(w_{ijl} | \mu_{ijl}^t, \tau_{ijl}^t) \quad (17.47)$$

where L is the number of layers, and D_l is the number of neurons in layer l . (The **expectation backpropagation** algorithm of [SHM14] is a special case of this, where the variances are fixed to $\tau = 1$.)

Suppose the parameters are static, so $\mathbf{w}_t = \mathbf{w}_{t-1}$. Then the new posterior, after conditioning on the t 'th observation, is given by

$$\hat{p}_t(\mathbf{w}) = \frac{1}{Z_t} p(\mathbf{y}_t | \mathbf{u}_t, \mathbf{w}) \mathcal{N}(\mathbf{w} | \boldsymbol{\mu}^{t-1}, \boldsymbol{\Sigma}^{t-1}) \quad (17.48)$$

where $\boldsymbol{\Sigma}^{t-1} = \text{diag}(\boldsymbol{\tau}^{t-1})$. We then project $\hat{p}_t(\mathbf{w})$ instead the space of factored Gaussians to compute the new (approximate) posterior, $p_t(\mathbf{w})$. This can be done by computing the following means and variances [Min01a]:

$$\mu_{ijl}^t = \mu_{ijl}^{t-1} + \tau_{ijl}^{t-1} \frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \quad (17.49)$$

$$\tau_{ijl}^t = \tau_{ijl}^{t-1} - (\tau_{ijl}^{t-1})^2 \left[\left(\frac{\partial \ln Z_t}{\partial \mu_{ijl}^{t-1}} \right)^2 - 2 \frac{\partial \ln Z_t}{\partial \tau_{ijl}^{t-1}} \right] \quad (17.50)$$

In the forwards pass, we compute Z_t by propagating the input \mathbf{u}_t through the model. Since we have a Gaussian distribution over the weights, instead of a point estimate, this induces an (approximately) Gaussian distribution over the values of the hidden units. For certain kinds of activation functions (such as ReLU), the relevant integrals (to compute the means and variances) can be solved analytically, as in GP-neural networks (Section 18.7). The result is that we get a Gaussian distribution over the final layer of the form $\mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\eta}_t = f(\mathbf{u}_t; \mathbf{w}_t)$ is the output of the neural network before the GLM link function induced by $p_t(\mathbf{w}_t)$. Hence we can approximate the partition function using

$$Z_t \approx \int p(\mathbf{y}_t | \boldsymbol{\eta}_t) \mathcal{N}(\boldsymbol{\eta}_t | \boldsymbol{\mu}, \boldsymbol{\Sigma}) d\boldsymbol{\eta}_t \quad (17.51)$$

We now discuss how to compute this integral. In the case of probit classification, with $y \in \{-1, +1\}$, we have $p(y|\mathbf{x}, \mathbf{w}) = \Phi(y\eta)$, where Φ is the cdf of the standard normal. We can then use the following analytical result

$$\int \Phi(y\eta) \mathcal{N}(h|\mu, \sigma) d\eta = \Phi\left(\frac{y\mu}{\sqrt{1+\sigma}}\right) \quad (17.52)$$

In the case of logistic classification, with $y \in \{0, 1\}$, we have $p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\sigma(\eta))$; in this case, we can use the probit approximation from Section 15.3.6. For the multiclass case, where $\mathbf{y} \in \{0, 1\}^C$ (one-hot encoding), we have $p(\mathbf{y}|\mathbf{x}, \mathbf{w}) = \text{Cat}(\mathbf{y}|\text{softmax}(\boldsymbol{\eta}))$. A variational lower bound to $\log Z_t$ for this case is given in [GDFY16].

Once we have computed Z_t , we can take gradients and update the Gaussian posterior moments, before moving to the next step.

17.5.4 Online variational inference for DNNs

A natural approach to online learning is to use variational inference, where the prior is the posterior from the previous step. This is known as **streaming variational Bayes** [Bro+13]. In more detail,

at step t , we compute

$$\psi_t = \operatorname{argmin}_{\psi} \underbrace{\mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + D_{\text{KL}}(q(\theta|\psi) \| q(\theta|\psi_{t-1}))}_{-\mathcal{L}_t(\psi)} \quad (17.53)$$

$$= \operatorname{argmin}_{\psi} \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta) + \log q(\theta|\psi) - \log q(\theta|\psi_{t-1})] \quad (17.54)$$

where $\ell_t(\theta) = -\log p(\mathcal{D}_t|\theta)$ is the negative log likelihood (or, more generally, some loss function) of the data batch at step t .

When applied to DNNs, this approach is called **variational continual learning** or **VCL** [Ngu+18]. (We discuss continual learning in Section 19.7.) An efficient implementation of this, known as **FOO-VB** (“fixed-point operator for online variational Bayes”) is given in [Zen+21].

One problem with the VCL objective in Equation (17.53) is that the KL term can cause the model to become too sparse, which can prevent the model from adapting or learning new tasks. This problem is called **variational overpruning** [TT17]. More precisely, the reason this happens as is as follows: some weights might not be needed to fit a given dataset, so their posterior will be equal to the prior, but sampling from these high-variance weights will add noise to the likelihood; to reduce this, the optimization method will prefer to set the bias term to a large negative value, so the corresponding unit is “turned off”, and thus has no effect on the likelihood. Unfortunately, these “dead units” become stuck, so there is not enough network capacity to learn the next task.

In [LST21], they propose a solution to this, known as **generalized variational continual learning** or **GVCL**. The first step is to downweight the KL term by a factor $\beta < 1$ to get

$$\mathcal{L}_t = \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + \beta D_{\text{KL}}(q(\theta|\psi) \| q(\theta|\psi_{t-1})) \quad (17.55)$$

Interestingly, one can show that in the limit of $\beta \rightarrow 0$, this recovers several standard methods that use a Laplace approximation based on the Hessian. In particular if we use a diagonal variational posterior, this reduces to online EWC method of [Sch+18]; if we use a block-diagonal and Kronecker factored posterior, this reduces to the online structured Laplace method of [RBB18b]; and if we use a low-rank posterior precision matrix, this reduces to the SOLA method of [Yin+20].

The second step is to replace the prior and posterior by using tempering, which is useful when the model is misspecified, as discussed in Section 17.3.11. In the case of Gaussians, raising the distribution to the power λ is equivalent to tempering with a temperature of $\tau = 1/\lambda$, which is the same as scaling the covariance by λ^{-1} . Thus the GVCL objective becomes

$$\mathcal{L}_t = \mathbb{E}_{q(\theta|\psi)} [\ell_t(\theta)] + \beta D_{\text{KL}}(q(\theta|\psi)^\lambda \| q(\theta|\psi_{t-1})^\lambda) \quad (17.56)$$

This can be optimized using SGD, assuming the posterior is reparameterizable (see Section 10.2.1).

17.6 Hierarchical Bayesian neural networks

In some problems, we have multiple related datasets, such as a set of medical images from different hospitals. Some aspects of the data (e.g., the shape of healthy vs diseased cells) is generally the same across datasets, but other aspects may be unique or idiosyncratic (e.g., each hospital may use a different colored die for staining). To model this, we can use a hierarchical Bayesian model, in which we allow the parameters for each dataset to be different (to capture random effects), while coming from

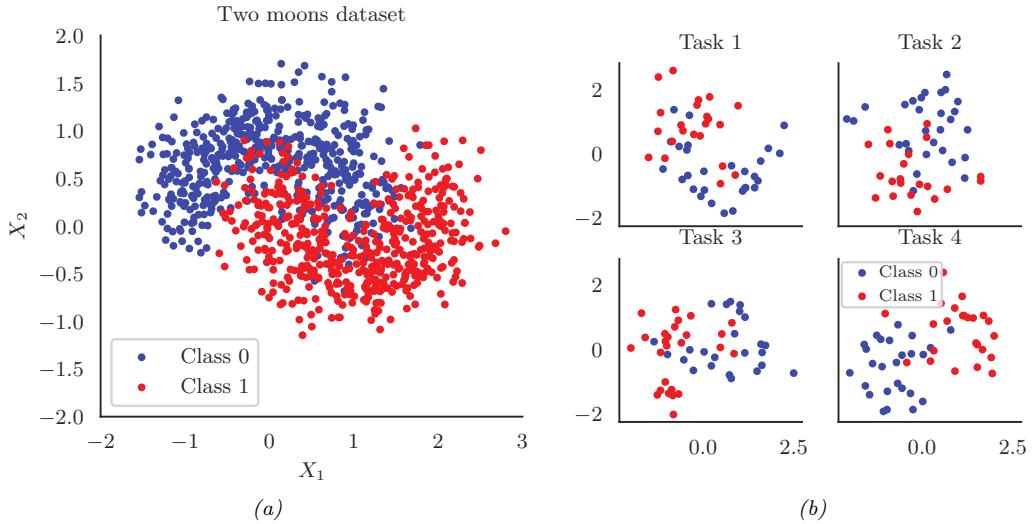


Figure 17.17: (a) Two moons synthetic dataset. (b) Multi-task version, where we rotate the data to create 18 related tasks (groups). Each dataset has 50 training and 50 test points. Here we show the first 4 tasks. Generated by [bnn_hierarchical.ipynb](#).

a common prior (to capture shared effects). This is the setup we considered in Section 15.5, where we discuss hierarchical Bayesian GLMs. In this section, we extend this to nonlinear predictors based on neural networks. (The setup is very similar to domain generalization, discussed in Section 19.6.2, except here we care about performance on all the domains, not just a held-out target domain.)

17.6.1 Example: multimoons classification

In this section, we consider an example² where we want to solve multiple related nonlinear binary classification problems coming from J different environments or distributions. We assume that each environment has its own unique decision boundary $p(y|\mathbf{x}, \mathbf{w}^j)$, so this is a form of concept shift (see Section 19.2.3). However we assume the overall shape of each boundary is similar to a common shared boundary, denote $p(y|\mathbf{x}, \mathbf{w}^0)$. We only have a small number N_j of examples from each environment, $\mathcal{D}^j = \{(\mathbf{x}_n^j, y_n^j) : n = 1 : N_j\}$, but we can utilize their common structure to do better than fitting J separate models.

To illustrate this, we create some synthetic 2d data for the $J = 18$ tasks. We start with the two-moons dataset, illustrated in Figure 17.17a. Each task is obtained by rotating the 2d inputs by a different amount, to create 18 related classification problems (see Figure 17.17b). See Figure 17.17b for the training data for 4 tasks.

To handle the nonlinear decision boundary, we use a multilayer perceptron. Since the dataset is low-dimensional (2d input), we use a shallow model with just 2 hidden layers, each with 5 neurons. We could fit a separate MLP to each task, but since we have limited data per task ($N_j = 50$ examples

2. This example is from https://tweicki.io/blog/2018/08/13/hierarchical_bayesian_neural_network/. For a real-world example of a similar approach applied to a gesture recognition task, see [Jos+17].

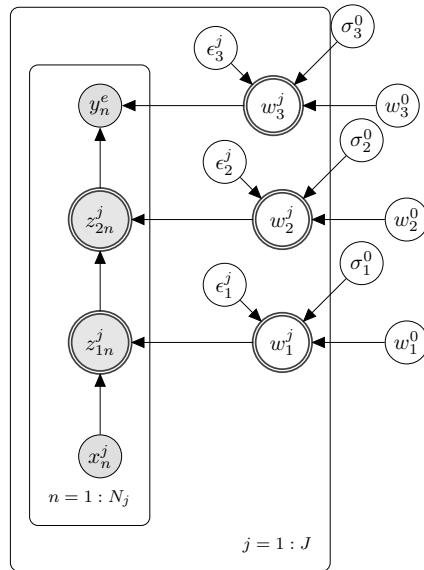


Figure 17.18: Illustration of a hierarchical Bayesian MLP with 2 hidden layers. There are J different models, each with N_j observed samples, and a common set of global shared parent parameters denoted with the 0 superscript. Nodes which are shaded are observed. Nodes with double ringed circles are deterministic functions of their parents.

for training), this works poorly, as we show below. We could also pool all the data and fit a single model, but this does even worse, since the datasets come from different underlying distributions, so mixing the data together from different ‘‘concepts’’ confuses the model. Instead we adopt a hierarchical Bayesian approach.

Our modeling assumptions are shown in Figure 17.18. In particular, we assume the weight from unit i to unit k in layer l for environment j , denoted $w_{i,k,l}^j$, comes from a common prior value $w_{i,k,l}^0$, with a random offset. We use the non-centered parameterization from Section 12.6.5 to write

$$w_{i,k,l}^j = w_{i,k,l}^0 + \epsilon_{i,k,l}^j \times \sigma_l^0 \quad (17.57)$$

where $\epsilon_{i,k,l}^j \sim \mathcal{N}(0, 1)$. By allowing a different σ_l^0 per layer l , we let the model control the degree of shrinkage to the prior for each layer separately. (We could also make the σ_l^j parameters be environment specific, which would allow for different amounts of distribution shift from the common parent.) For the hyper-parameters, we put $\mathcal{N}(0, 1)$ priors on $w_{i,k,l}^0$, and $\mathcal{N}_+(1)$ priors on σ_l^0 .

We compute the posterior $p(\epsilon_{1:L}^{1:J}, w_{1:L}^0, \sigma_{1:L}^0 | \mathcal{D})$ using HMC (Section 12.5). We then evaluate this model using a fresh set of labeled samples from each environment. The average classification accuracy on the train and test sets for the non-hierarchical model (one MLP per environment, fit separately) is 86% and 83%. For the hierarchical model, this improves to 91% and 89% respectively.

To see why the hierarchical model works better, we will plot the posterior predictive distribution in 2d. Figure 17.19(top) shows the results for the nonhierarchical models; we see that the method

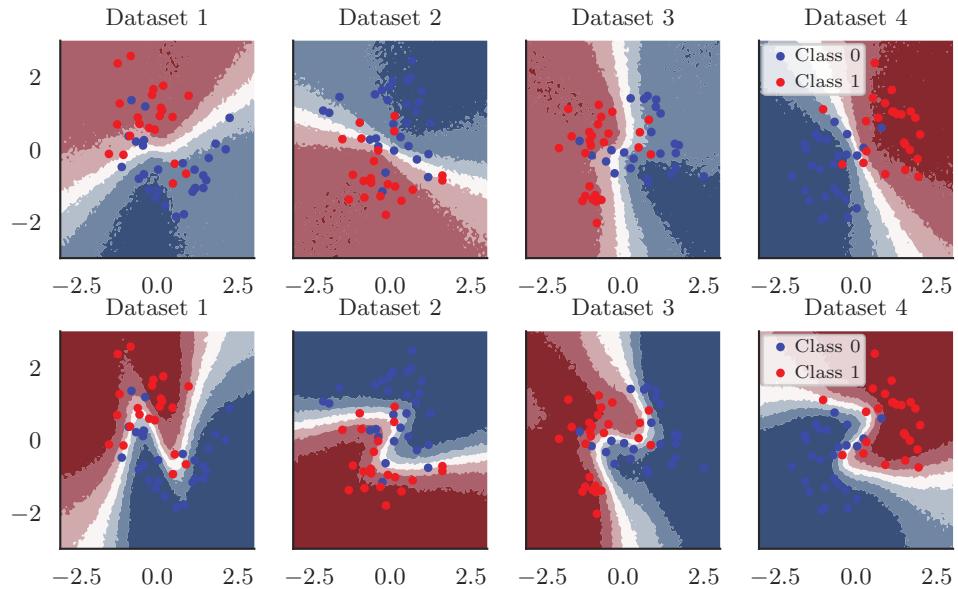


Figure 17.19: Top: Results of fitting separate MLPs on each dataset. Bottom: Results of fitting hierarchical MLP on all datasets jointly. Generated by [bnm_hierarchical.ipynb](#).

fails to learn the common underlying Z-shaped decision boundary. By contrast, Figure 17.19(bottom) shows that the hierarchical method has correctly recovered the common pattern, while still allowing group variation.

18 Gaussian processes

This chapter is coauthored with Andrew Wilson.

18.1 Introduction

Deep neural networks are a family of flexible function approximators of the form $f(\mathbf{x}; \boldsymbol{\theta})$, where the dimensionality of $\boldsymbol{\theta}$ (i.e., the number of parameters) is fixed, and independent of the size N of the training set. However, such parametric models can overfit when N is small, and can underfit when N is large, due to their fixed capacity. In order to create models whose capacity automatically adapts to the amount of data, we turn to **nonparametric models**.

There are many approaches to building nonparametric models for classification and regression (see e.g., [Was06]). In this chapter, we consider a Bayesian approach in which we represent uncertainty about the input-output mapping f by defining a prior distribution over functions, and then updating it given data. In particular, we will use a **Gaussian process** to represent the prior $p(f)$; we then use Bayes' rule to derive the posterior $p(f|\mathcal{D})$, which is another GP, as we explain below. More details on GPs can be found the excellent book [RW06], as well as the interative tutorial at <https://distill.pub/2019/visual-exploration-gaussian-processes>. See also Chapter 31 for other examples of Bayesian nonparametric models.

18.1.1 GPs: what and why?

To explain GPs in more detail, recall that a Gaussian random vector of length N , $\mathbf{f} = [f_1, \dots, f_N]$, is defined by its mean $\boldsymbol{\mu} = \mathbb{E}[\mathbf{f}]$ and its covariance $\boldsymbol{\Sigma} = \text{Cov}[\mathbf{f}]$. Now consider a function $f : \mathcal{X} \rightarrow \mathbb{R}$ evaluated at a set of inputs, $\mathbf{X} = \{\mathbf{x}_n \in \mathcal{X}\}_{n=1}^N$. Let $\mathbf{f}_X = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_N)]$ be the set of unknown function values at these points. If \mathbf{f}_X is jointly Gaussian for any set of $N \geq 1$ points, then we say that $f : \mathcal{X} \rightarrow \mathbb{R}$ is a **Gaussian process**. Such a process is defined by its **mean function** $m(\mathbf{x}) \in \mathbb{R}$ and a **covariance function**, $\mathcal{K}(\mathbf{x}, \mathbf{x}') \geq 0$, which is any positive definite **Mercer kernel** (see Section 18.2). For example, we might use an RBF kernel of the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') \propto \exp(-\|\mathbf{x} - \mathbf{x}'\|^2)$ (see Section 18.2.1.1 for details).

We denote the corresponding GP by

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), \mathcal{K}(\mathbf{x}, \mathbf{x}')) \tag{18.1}$$

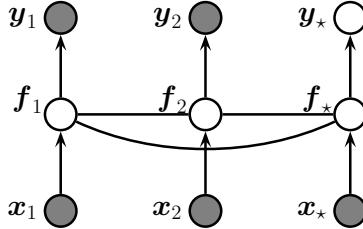


Figure 18.1: A Gaussian process for 2 training points, \mathbf{x}_1 and \mathbf{x}_2 , and 1 testing point, \mathbf{x}_* , represented as a graphical model representing $p(\mathbf{y}, \mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | m(\mathbf{X}), \mathcal{K}(\mathbf{X})) \prod_i p(y_i | f_i)$. The hidden nodes $f_i = f(\mathbf{x}_i)$ represent the value of the function at each of the datapoints. These hidden nodes are fully interconnected by undirected edges, forming a Gaussian graphical model; the edge strengths represent the covariance terms $\Sigma_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$. If the test point \mathbf{x}_* is similar to the training points \mathbf{x}_1 and \mathbf{x}_2 , then the value of the hidden function f_* will be similar to f_1 and f_2 , and hence the predicted output y_* will be similar to the training values y_1 and y_2 .

where

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \quad (18.2)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))^\top] \quad (18.3)$$

This means that, for any finite set of points $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, we have

$$p(\mathbf{f}_X | \mathbf{X}) = \mathcal{N}(\mathbf{f}_X | \boldsymbol{\mu}_X, \mathbf{K}_{X,X}) \quad (18.4)$$

where $\boldsymbol{\mu}_X = (m(\mathbf{x}_1), \dots, m(\mathbf{x}_N))$ and $\mathbf{K}_{X,X}(i, j) \triangleq \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$.

A GP can be used to define a prior over functions. We can evaluate this prior at any set of points we choose. However, to learn about the function from data, we have to update this prior with a likelihood function. We typically assume we have a set of N iid observations $\mathcal{D} = \{(\mathbf{x}_i, y_i) : i = 1 : N\}$, where $y_i \sim p(y | f(\mathbf{x}_i))$, as shown in Figure 18.1. If we use a Gaussian likelihood, we can compute the posterior $p(f | \mathcal{D})$ in closed form, as we discuss in Section 18.3. For other kinds of likelihoods, we will need to use approximate inference, as we discuss in Section 18.4. In many cases f is not directly observed, and instead forms part of a latent variable model, both in supervised and unsupervised settings such as in Section 28.3.7.

The generalization properties of a Gaussian process are controlled by its covariance function (kernel), which we describe in Section 18.2. These kernels live in a reproducing kernel Hilbert space (RKHS), described in Section 18.3.7.1.

GPs were originally designed for spatial data analysis, where the input is 2d. This special case is called **kriging**. However, they can be applied to higher dimensional inputs. In addition, while they have been traditionally limited to small datasets, it is now possible to apply GPs to problems with millions of points, with essentially exact inference. We discuss these scalability advances in Section 18.5.

Moreover, while Gaussian processes have historically been considered smoothing interpolators, GPs now routinely perform representation learning, through covariance function learning, and multilayer

models. These advances have clearly illustrated that GPs and neural networks are not competing, but complementary, and can be combined for better performance than would be achieved by deep learning alone. We describe GPs for representation learning in Section 18.6.

The connections between Gaussian processes and neural networks can also be further understood by considering infinite limits of neural networks that converge to Gaussian processes with particular covariance functions, which we describe in Section 18.7.

So Gaussian processes are nonparametric models which can scale and do representation learning. But why, in the age of deep learning, should we want to use a Gaussian process? There are several compelling reasons to prefer a GP, including:

- Gaussian processes typically provide well-calibrated predictive distributions, with a good characterization of epistemic (model) uncertainty — uncertainty arising from not knowing which of many solutions is correct. For example, as we move away from the data, there are a greater variety of consistent solutions, and so we expect greater uncertainty.
- Gaussian processes are often state-of-the-art for continuous regression problems, especially spatiotemporal problems, such as weather interpolation and forecasting. In regression, Gaussian process inference can also typically be performed in closed form.
- The marginal likelihood of a Gaussian process provides a powerful mechanism for flexible kernel learning. Kernel learning enables us to provide long-range extrapolations, but also tells us interpretable properties of the data that we didn't know before, towards scientific discovery.
- Gaussian processes are often used as a probabilistic surrogate for optimizing expensive objectives, in a procedure known as **Bayesian optimization** (Section 6.6).

18.2 Mercer kernels

The generalization properties of Gaussian processes boil down to how we encode prior knowledge about the similarity of two input vectors. If we know that \mathbf{x}_i is similar to \mathbf{x}_j , then we can encourage the model to make the predicted output at both locations (i.e., $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$) to be similar.

To define similarity, we introduce the notion of a **kernel function**. The word “kernel” has many different meanings in mathematics; here we consider a **Mercer kernel**, also called a **positive definite kernel**. This is any symmetric function $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$ such that

$$\sum_{i=1}^N \sum_{j=1}^N \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) c_i c_j \geq 0 \quad (18.5)$$

for any set of N (unique) points $\mathbf{x}_i \in \mathcal{X}$, and any choice of numbers $c_i \in \mathbb{R}$. We assume $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) > 0$, so that we can only achieve equality in the above equation if $c_i = 0$ for all i .

Another way to understand this condition is the following. Given a set of N datapoints, let us define the **Gram matrix** as the following $N \times N$ similarity matrix:

$$\mathbf{K} = \begin{pmatrix} \mathcal{K}(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_1, \mathbf{x}_N) \\ & \vdots & \\ \mathcal{K}(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \mathcal{K}(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} \quad (18.6)$$

We say that \mathcal{K} is a Mercer kernel iff the Gram matrix is positive definite for any set of (distinct) inputs $\{\mathbf{x}_i\}_{i=1}^N$.

We discuss several popular Mercer kernels below. More details can be found at [Wil14] and <https://www.cs.toronto.edu/~duvenaud/cookbook/>. See also Section 18.6 where we discuss how to learn kernels from data.

18.2.1 Stationary kernels

For real-valued inputs, $\mathcal{X} = \mathbb{R}^D$, it is common to use **stationary kernels** (also called **shift-invariant kernels**), which are functions of the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}(\mathbf{r})$, where $\mathbf{r} = \mathbf{x} - \mathbf{x}'$; thus the output only depends on the relative difference between the inputs. (See Section 18.2.2 for a discussion of non-stationary kernels.) Furthermore, in many cases, all that matters is the magnitude of the difference:

$$r = \|\mathbf{r}\|_2 = \|\mathbf{x} - \mathbf{x}'\| \quad (18.7)$$

We give some examples below. (See also Figure 18.3 and Figure 18.4 for some visualizations of these kernels.)

18.2.1.1 Squared exponential (RBF) kernel

The **squared exponential** (SE) kernel, also sometimes called the **exponentiated quadratic** kernel or the **radial basis function** (RBF) kernel, is defined as

$$\mathcal{K}(r; \ell) = \exp\left(-\frac{r^2}{2\ell^2}\right) \quad (18.8)$$

Here ℓ corresponds to the **length-scale** of the kernel, i.e., the distance over which we expect differences to matter.

From Equation (18.7) we can rewrite this kernel as

$$\mathcal{K}(\mathbf{x}, \mathbf{x}'; \ell) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \quad (18.9)$$

This is the RBF kernel we encountered earlier. It is also sometimes called the **Gaussian kernel**.

See Figure 18.3(f) and Figure 18.4(f) for a visualization in 1D.

18.2.1.2 ARD kernel

We can generalize the RBF kernel by replacing Euclidean distance with Mahalanobis distance, as follows:

$$\mathcal{K}(\mathbf{r}; \boldsymbol{\Sigma}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \mathbf{r}^\top \boldsymbol{\Sigma}^{-1} \mathbf{r}\right) \quad (18.10)$$

where $\mathbf{r} = \mathbf{x} - \mathbf{x}'$. If $\boldsymbol{\Sigma}$ is diagonal, this can be written as

$$\mathcal{K}(\mathbf{r}; \boldsymbol{\ell}, \sigma^2) = \sigma^2 \exp\left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} r_d^2\right) = \prod_{d=1}^D \mathcal{K}(r_d; \ell_d, \sigma^{2/d}) \quad (18.11)$$

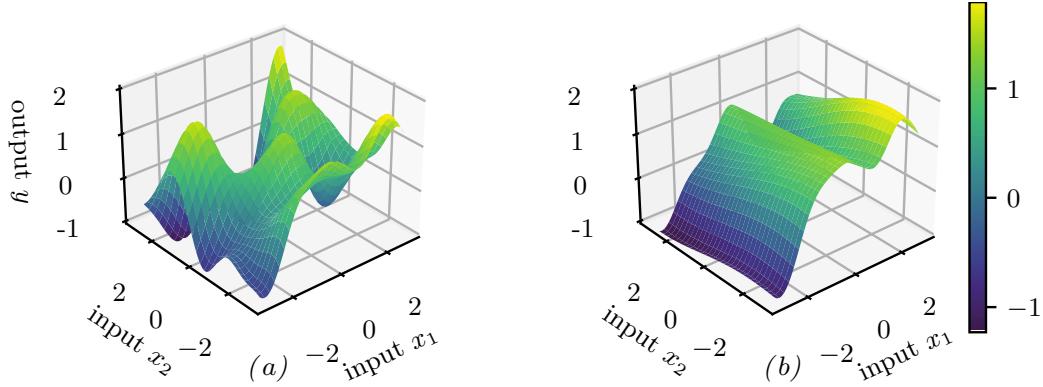


Figure 18.2: Function samples from a GP with an ARD kernel. (a) $\ell_1 = \ell_2 = 1$. Both dimensions contribute to the response. (b) $\ell_1 = 1$, $\ell_2 = 5$. The second dimension is essentially ignored. Adapted from Figure 5.1 of [RW06]. Generated by [gpr_demo_ard.ipynb](#).

where

$$\mathcal{K}(r; \ell, \tau^2) = \tau^2 \exp\left(-\frac{1}{2} \frac{1}{\ell^2} r^2\right) \quad (18.12)$$

We can interpret σ^2 as the overall variance, and ℓ_d as defining the **characteristic length scale** of dimension d . If d is an irrelevant input dimension, we can set $\ell_d = \infty$, so the corresponding dimension will be ignored. This is known as **automatic relevance determination** or **ARD** (Section 15.2.8). Hence the corresponding kernel is called the **ARD kernel**. See Figure 18.2 for an illustration of some 2d functions sampled from a GP using this prior.

18.2.1.3 Matérn kernels

The SE kernel gives rise to functions that are infinitely differentiable, and therefore are very smooth. For many applications, it is better to use the **Matérn kernel**, which gives rise to “rougher” functions, which can better model local “wiggles” without having to make the overall length scale very small.

The Matérn kernel has the following form:

$$\mathcal{K}(r; \nu, \ell) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu}r}{\ell} \right) \quad (18.13)$$

where K_ν is a modified Bessel function and ℓ is the length scale. Functions sampled from this GP are k -times differentiable iff $\nu > k$. As $\nu \rightarrow \infty$, this approaches the SE kernel.

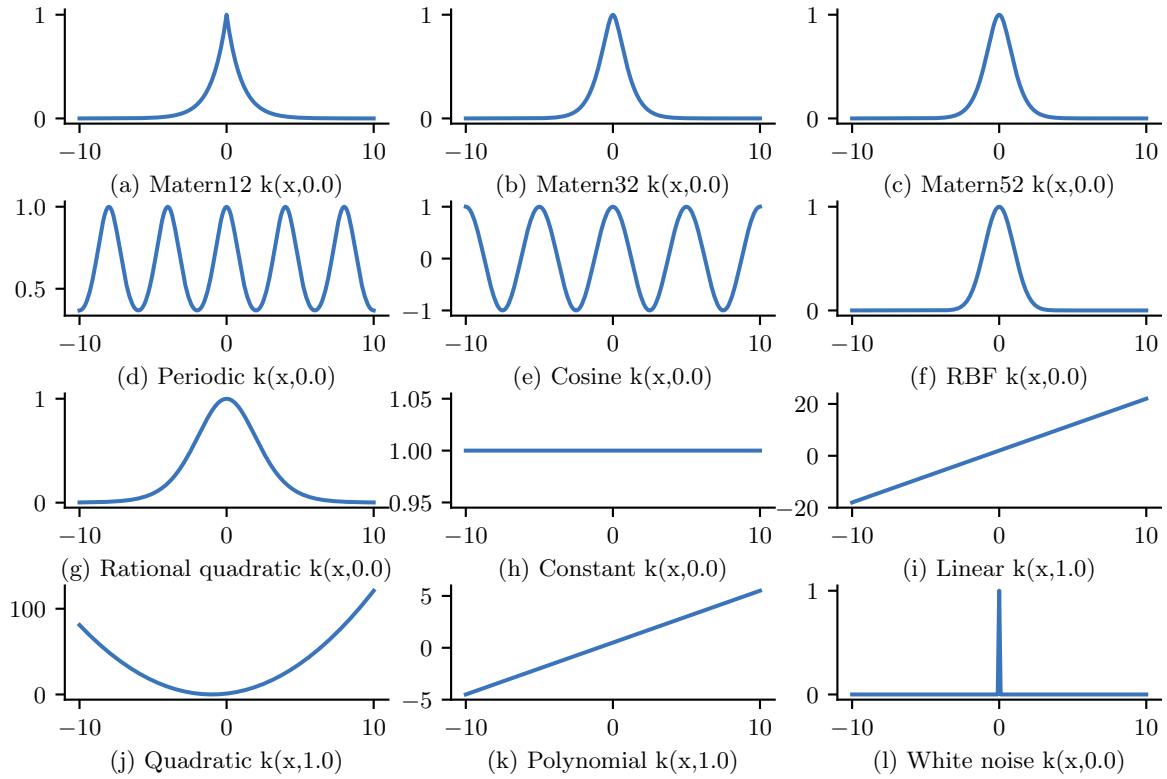


Figure 18.3: GP kernels evaluated at $k(x, 0)$ as a function of x . Generated by [gpKernelPlot.ipynb](#).

For values $\nu \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}\}$, the function simplifies as follows:

$$\mathcal{K}(r; \frac{1}{2}, \ell) = \exp\left(-\frac{r}{\ell}\right) \quad (18.14)$$

$$\mathcal{K}(r; \frac{3}{2}, \ell) = \left(1 + \frac{\sqrt{3}r}{\ell}\right) \exp\left(-\frac{\sqrt{3}r}{\ell}\right) \quad (18.15)$$

$$\mathcal{K}(r; \frac{5}{2}, \ell) = \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}r}{\ell}\right) \quad (18.16)$$

See Figure 18.3(a-c) and Figure 18.4(a-c) for a visualization.

The value $\nu = \frac{1}{2}$ corresponds to the **Ornstein-Uhlenbeck process**, which describes the velocity of a particle undergoing Brownian motion. The corresponding function is continuous but not differentiable, and hence is very “jagged”.

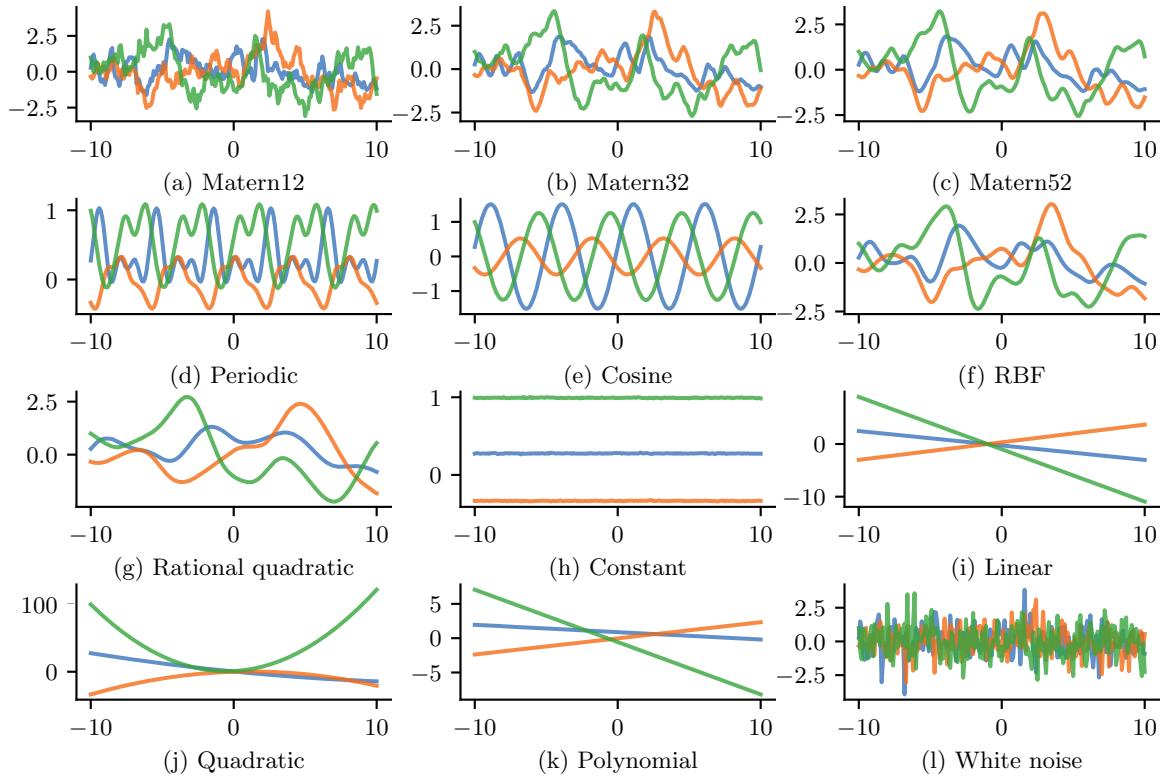


Figure 18.4: GP samples drawn using different kernels. Generated by [gpKernelPlot.ipynb](#).

18.2.1.4 Periodic kernels

One way to create a periodic 1d random function is to map x to the 2d space $\mathbf{u}(x) = (\cos(x), \sin(x))$, and then use an SE kernel in \mathbf{u} -space:

$$\mathcal{K}(x, x') = \exp\left(-\frac{2 \sin^2((x-x')/2)}{\ell^2}\right) \quad (18.17)$$

which follows since $(\cos(x) - \cos(x'))^2 + (\sin(x) - \sin(x'))^2 = 4 \sin^2((x-x')/2)$. We can generalize this by specifying the period p to get the **periodic kernel**, also called the **exp-sine-squared kernel**:

$$\mathcal{K}_{\text{per}}(r; \ell, p) = \exp\left(-\frac{2}{\ell^2} \sin^2(\pi \frac{r}{p})\right) \quad (18.18)$$

where p is the period and ℓ is the length scale. See Figure 18.3(d-e) and Figure 18.4(d-e) for a visualization.

A related kernel is the **cosine kernel**:

$$\mathcal{K}(r; p) = \cos\left(2\pi \frac{r}{p}\right) \quad (18.19)$$

18.2.1.5 Rational quadratic kernel

We define the **rational quadratic** kernel to be

$$\mathcal{K}_{RQ}(r; \ell, \alpha) = \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha} \quad (18.20)$$

We recognize this is proportional to a Student t density. Hence it can be interpreted as a scale mixture of SE kernels of different characteristic lengths. In particular, let $\tau = 1/\ell^2$, and assume $\tau \sim \text{Ga}(\alpha, \ell^2)$. Then one can show that

$$\mathcal{K}_{RQ}(r) = \int p(\tau|\alpha, \ell^2) \mathcal{K}_{SE}(r|\tau) d\tau \quad (18.21)$$

As $\alpha \rightarrow \infty$, this reduces to a SE kernel.

See Figure 18.3(g) and Figure 18.4(g) for a visualization.

18.2.1.6 Kernels from spectral densities

Consider the case of a stationary kernel which satisfies $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}(\boldsymbol{\delta})$, where $\boldsymbol{\delta} = \mathbf{x} - \mathbf{x}'$, for $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$. Let us further assume that $\mathcal{K}(\boldsymbol{\delta})$ is positive definite. In this case, **Bochner's theorem** tells us that we can represent $\mathcal{K}(\boldsymbol{\delta})$ by its Fourier transform:

$$\mathcal{K}(\boldsymbol{\delta}) = \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^\top \boldsymbol{\delta}} d\boldsymbol{\omega} \quad (18.22)$$

where $j = \sqrt{-1}$, $e^{j\theta} = \cos(\theta) + j \sin(\theta)$, $\boldsymbol{\omega}$ is the frequency, and $p(\boldsymbol{\omega})$ is the **spectral density** (see [SS19, p93, p253] for details).

We can easily derive and gain intuitions into several kernels from spectral densities. If we take the Fourier transform of an RBF kernel we find the spectral density $p(\boldsymbol{\omega}) = \sqrt{2\pi\ell^2} \exp(-2\pi^2\boldsymbol{\omega}^2\ell^2)$. Thus the spectral density is also Gaussian, but with a bandwidth *inversely* proportional to the length-scale hyperparameter ℓ . That is, as ℓ becomes large, the spectral density collapses onto a point mass. This result is intuitive: as we increase the length-scale, our model treats points as correlated over large distances, and becomes very smooth and slowly varying, and thus low-frequency. In general, since the Gaussian distribution has relatively light tails, we can see that RBF kernels won't generally support high frequency solutions.

We can instead use a Student t spectral density, which has heavy tails that will provide greater support for higher frequencies. Taking the inverse Fourier transform of this spectral density, we recover the Matérn kernel, with degrees of freedom ν corresponding to the degrees of freedom in the spectral density. Indeed, the smaller we make ν , the less smooth and higher frequency are the associated fits to data using a Matérn kernel.

We can also derive **spectral mixture kernels** by modelling the spectral density as a scale-location mixture of Gaussians and taking the inverse Fourier transform [WA13]. Since scale-location mixtures of Gaussians are dense in the set of distributions, and can therefore approximate any spectral density, this kernel can approximate any stationary kernel to arbitrary precision. The spectral mixture kernel thus forms a powerful approach to kernel learning, which we discuss further in Section 18.6.5.

18.2.2 Nonstationary kernels

A stationary kernel assumes the measure of similarity between two inputs is independent of their location, i.e., $\mathcal{K}(\mathbf{x}, \mathbf{x}')$ only depends on $\mathbf{r} = \mathbf{x} - \mathbf{x}'$. A **nonstationary kernel** relaxes this assumption. This is useful for a variety of problems, such as environmental modeling (see e.g., [GSR12; Pat+22]), where correlations between locations can change depending on latent factors in the environment.

18.2.2.1 Polynomial kernels

A simple form of non-stationary kernel is the **polynomial kernel** (also called **dot product kernel**) of order M , defined by

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}')^M \quad (18.23)$$

This contains all monomials of order M . For example, if $M = 2$, we get the **quadratic kernel**; in 2d, this becomes

$$(\mathbf{x}^\top \mathbf{x}')^2 = (x_1 x'_1 + x_2 x'_2)^2 = (x_1 x'_1)^2 + (x_2 x'_2)^2 + 2(x_1 x'_1)(x_2 x'_2) \quad (18.24)$$

We can generalize this to contain all terms up to degree M by using the **inhomogeneous polynomial kernel**

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + c)^M \quad (18.25)$$

For example, if $M = 2$ and the inputs are 2d, we have

$$\begin{aligned} (\mathbf{x}^\top \mathbf{x}' + 1)^2 &= (x_1 x'_1)^2 + (x_1 x'_1)(x_2 x'_2) + (x_1 x'_1) \\ &\quad + (x_2 x'_2)(x_1 x'_1) + (x_2 x'_2)^2 + (x_2 x'_2) \\ &\quad + (x_1 x'_1) + (x_2 x'_2) + 1 \end{aligned} \quad (18.26)$$

18.2.2.2 Gibbs kernel

Consider an RBF kernel where the length scale hyper-parameter, and the signal variance hyper-parameter, are both input dependent; this is called the **Gibbs kernel** [Gib97], and is defined by

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sigma(\mathbf{x})\sigma(\mathbf{x}') \sqrt{\frac{2\ell(\mathbf{x})\ell(\mathbf{x}')}{\ell(\mathbf{x})^2 + \ell(\mathbf{x}')^2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{\ell(\mathbf{x})^2 + \ell(\mathbf{x}')^2}\right) \quad (18.27)$$

If $\ell(\mathbf{x})$ and $\sigma(\mathbf{x})$ are constants, this reduces to the standard RBF kernel. We can model the functional dependency of these kernel parameters on the input by using another GP (see e.g., [Hei+16]).

18.2.2.3 Other non-stationary kernels

Other ways to induce non-stationarity include using a neural network kernel (Section 18.7.1), non-stationary spectral kernels [RHK17], or a deep GP (Section 18.7.3).

18.2.3 Kernels for nonvectorial (structured) inputs

Kernels are particularly useful when the inputs are structured objects, such as strings and graphs, since it is often hard to “featurize” variable-sized inputs. For example, we can define a **string kernel** which compares strings in terms of the number of n-grams they have in common [Lod+02; BC17].

We can also define kernels on graphs [KJM19]. For example, the **random walk kernel** conceptually performs random walks on two graphs simultaneously, and then counts the number of paths that were produced by both walks. This can be computed efficiently as discussed in [Vis+10]. For more details on graph kernels, see [KJM19].

For a review of kernels on structured objects, see e.g., [Gär03].

18.2.4 Making new kernels from old

Given two valid kernels $\mathcal{K}_1(\mathbf{x}, \mathbf{x}')$ and $\mathcal{K}_2(\mathbf{x}, \mathbf{x}')$, we can create a new kernel using any of the following methods:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = c\mathcal{K}_1(\mathbf{x}, \mathbf{x}'), \text{ for any constant } c > 0 \quad (18.28)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})\mathcal{K}_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}'), \text{ for any function } f \quad (18.29)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = q(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \text{ for any function polynomial } q \text{ with nonneg. coef.} \quad (18.30)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(\mathcal{K}_1(\mathbf{x}, \mathbf{x}')) \quad (18.31)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{A} \mathbf{x}', \text{ for any psd matrix } \mathbf{A} \quad (18.32)$$

For example, suppose we start with the linear kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathbf{x}\mathbf{x}'$. We know this is a valid Mercer kernel, since the corresponding Gram matrix is just the (scaled) covariance matrix of the data. From the above rules, we can see that the polynomial kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}')^M$ from Section 18.2.2.1 is a valid Mercer kernel.

We can also use the above rules to establish that the Gaussian kernel is a valid kernel. To see this, note that

$$\|\mathbf{x} - \mathbf{x}'\|^2 = \mathbf{x}^\top \mathbf{x} + (\mathbf{x}')^\top \mathbf{x}' - 2\mathbf{x}^\top \mathbf{x}' \quad (18.33)$$

and hence

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2) = \exp(-\mathbf{x}^\top \mathbf{x} / 2\sigma^2) \exp(\mathbf{x}^\top \mathbf{x}' / \sigma^2) \exp(-(\mathbf{x}')^\top \mathbf{x}' / 2\sigma^2) \quad (18.34)$$

is a valid kernel.

We can also combine kernels using addition or multiplication:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') + \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.35)$$

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \mathcal{K}_1(\mathbf{x}, \mathbf{x}') \times \mathcal{K}_2(\mathbf{x}, \mathbf{x}') \quad (18.36)$$

Multiplying two positive-definite kernels together always results in another positive definite kernel. This is a way to get a conjunction of the individual properties of each kernel, as illustrated in Figure 18.5.

In addition, adding two positive-definite kernels together always results in another positive definite kernel. This is a way to get a disjunction of the individual properties of each kernel, as illustrated in Figure 18.6.

For an example of combining kernels to forecast some time series data, see Section 18.8.1.

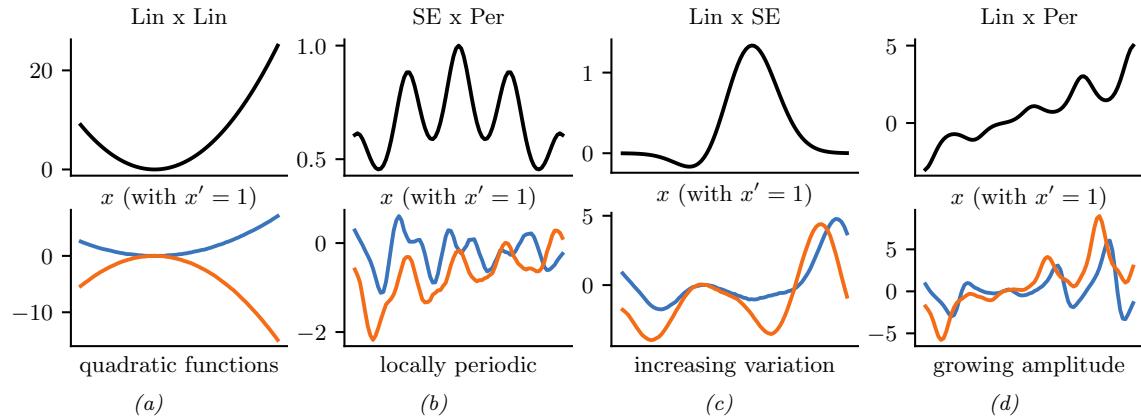


Figure 18.5: Examples of 1d structures obtained by multiplying elementary kernels. Top row shows $K(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, \mathcal{K})$. Adapted from Figure 2.2 of [Duv14]. Generated by [combining_kernels_by_multiplication.ipynb](#).

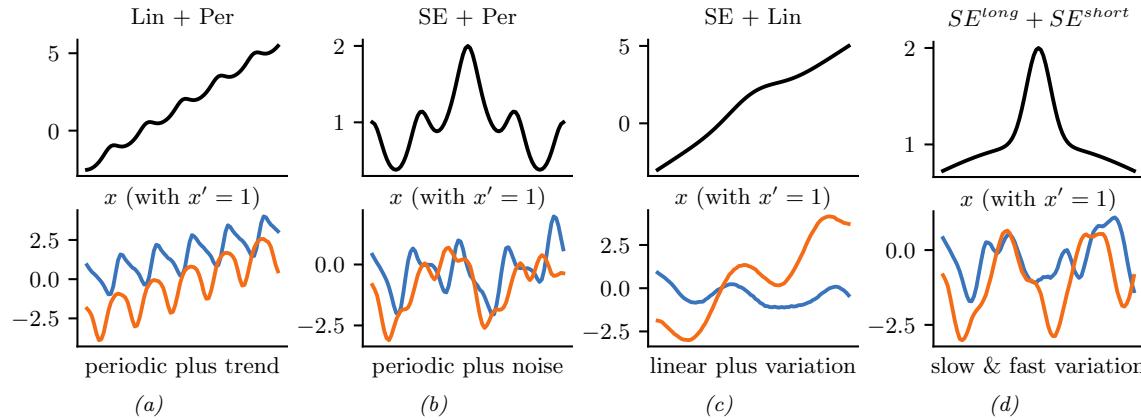


Figure 18.6: Examples of 1d structures obtained by summing elementary kernels. Top row shows $K(x, x' = 1)$. Bottom row shows some functions sampled from $GP(f|0, \mathcal{K})$. Adapted from Figure 2.2 of [Duv14]. Generated by [combining_kernels_by_summation.ipynb](#).

18.2.5 Mercer's theorem

Recall that any positive definite matrix \mathbf{K} can be represented using an eigendecomposition of the form $\mathbf{K} = \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$, and \mathbf{U} is a matrix containing the eigenvectors. Now consider element (i, j) of \mathbf{K} :

$$k_{ij} = (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:i})^\top (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:j}) \quad (18.37)$$

where $\mathbf{U}_{:i}$ is the i 'th column of \mathbf{U} . If we define $\phi(\mathbf{x}_i) = \mathbf{U}_{:i}$, then we can write

$$k_{ij} = \sum_{m=1}^M \lambda_m \phi_m(\mathbf{x}_i) \phi_m(\mathbf{x}_j) \quad (18.38)$$

where M is the rank of the kernel matrix. Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors of the kernel matrix.

This idea can be generalized to apply to kernel functions, not just kernel matrices, as we now show. First, we define an **eigenfunction** $\phi()$ of a kernel \mathcal{K} with eigenvalue λ wrt measure μ as a function that satisfies

$$\int \mathcal{K}(\mathbf{x}, \mathbf{x}') \phi(\mathbf{x}) d\mu(\mathbf{x}) = \lambda \phi(\mathbf{x}') \quad (18.39)$$

We usually sort the eigenfunctions in order of decreasing eigenvalue, $\lambda_1 \geq \lambda_2 \geq \dots$. The eigenfunctions are orthogonal wrt μ :

$$\int \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mu(\mathbf{x}) = \delta_{ij} \quad (18.40)$$

where δ_{ij} is the Kronecker delta. With this definition in hand, we can state **Mercer's theorem**. Informally, it says that any positive definite kernel function can be represented as the following infinite sum:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{m=1}^{\infty} \lambda_m \phi_m(\mathbf{x}) \phi_m(\mathbf{x}') \quad (18.41)$$

where ϕ_m are eigenfunctions of the kernel, and λ_m are the corresponding eigenvalues. This is the functional analog of Equation (18.38).

A **degenerate kernel** has only a finite number of non-zero eigenvalues. In this case, we can rewrite the kernel function as an inner product between two finite-length vectors. For example, consider the quadratic kernel $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle^2$ from Equation (18.24). If we define $\phi(x_1, x_2) = [x_1^2, \sqrt{2}x_1x_2, x_2^2] \in \mathbb{R}^3$, then we can write this as $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \phi(\mathbf{x}')$. Thus we see that this kernel is degenerate.

Now consider the RBF kernel. In this case, the corresponding feature representation is infinite dimensional (see Section 18.2.6 for details). However, by working with kernel functions, we can avoid having to deal with infinite dimensional vectors.

From the above, we see that we can replace inner product operations in an explicit (possibly infinite dimensional) feature space with a call to a kernel function, i.e., we replace $\phi(\mathbf{x})^\top \phi(\mathbf{x})$ with $\mathcal{K}(\mathbf{x}, \mathbf{x}')$. This is called the **kernel trick**.

18.2.6 Approximating kernels with random features

Although the power of kernels resides in the ability to avoid working with featurized representations of the inputs, such kernelized methods can take $O(N^3)$ time, in order to invert the Gram matrix \mathbf{K} , as we will see in Section 18.3. This can make it difficult to use such methods on large scale data.

Fortunately, we can approximate the feature map for many kernels using a randomly chosen finite set of M basis functions, thus reducing the cost to $O(NM + M^3)$.

We will show how to do this for shift-invariant kernels by returning to Bochner's theorem in Eq. (18.22). In the case of a Gaussian RBF kernel, we have seen that the spectral density is a Gaussian distribution. Hence we can easily compute a Monte Carlo approximation to this integral by sampling random Gaussian vectors. This yields the following approximation: $\mathcal{K}(\mathbf{x}, \mathbf{x}') \approx \phi(\mathbf{x})^\top \phi(\mathbf{x}')$, where the (real-valued) feature vector is given by

$$\phi(\mathbf{x}) = \sqrt{\frac{1}{D}} [\sin(\mathbf{z}_1^\top \mathbf{x}), \dots, \sin(\mathbf{z}_D^\top \mathbf{x}), \cos(\mathbf{z}_1^\top \mathbf{x}), \dots, \cos(\mathbf{z}_D^\top \mathbf{x})] \quad (18.42)$$

$$= \sqrt{\frac{1}{D}} [\sin(\mathbf{Z}^\top \mathbf{x}), \cos(\mathbf{Z}^\top \mathbf{x})] \quad (18.43)$$

Here $\mathbf{Z} = (1/\sigma)\mathbf{G}$, and $\mathbf{G} \in \mathbb{R}^{d \times D}$ is a random Gaussian matrix, where the entries are sampled iid from $\mathcal{N}(0, 1)$. The representation in Equation (18.43) are called **random Fourier features (RFF)** [SS15; RR08] or “weighted sums of random kitchen sinks” [RR09]. (One can obtain an even better approximation by ensuring that the rows of \mathbf{Z} are random but orthogonal; this is called **orthogonal random features** [Yu+16].)

One can create similar random feature representations for other kinds of kernels. We can then use such features for supervised learning by defining $f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{W}\varphi(\mathbf{Z}\mathbf{x}) + \mathbf{b}$, where \mathbf{Z} is a random Gaussian matrix, and the form of φ depends on the chosen kernel. This is equivalent to a one layer MLP with random input-to-hidden weights; since we only optimize the hidden-to-output weights $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$, the model is equivalent to a linear model with fixed random features. If we use enough random features, we can approximate the performance of a kernelized prediction model, but the computational cost is now $O(N)$ rather than $O(N^2)$.

Unfortunately, random features can result in worse performance than using a non-degenerate kernel, since they don't have enough expressive power. We discuss other ways to scale GPs to large datasets in Section 18.5.

18.3 GPs with Gaussian likelihoods

In this section, we discuss GPs for regression, using a Gaussian likelihood. In this case, all the computations can be performed in closed form, using standard linear algebra methods. We extend this framework to non-Gaussian likelihoods later in the chapter.

18.3.1 Predictions using noise-free observations

Suppose we observe a training set $\mathcal{D} = \{(\mathbf{x}_n, y_n) : n = 1 : N\}$, where $y_n = f(\mathbf{x}_n)$ is the noise-free observation of the function evaluated at \mathbf{x}_n . If we ask the GP to predict $f(\mathbf{x})$ for a value of \mathbf{x} that it has already seen, we want the GP to return the answer $f(\mathbf{x})$ with no uncertainty. In other words, it should act as an **interpolator** of the training data. Here we assume the observed function values are noiseless. We will consider the case of noisy observations shortly.

Now we consider the case of predicting the outputs for new inputs that may not be in \mathcal{D} . Specifically, given a test set \mathbf{X}_* of size $N_* \times D$, we want to predict the function outputs $\mathbf{f}_* = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_{N_*})]$.

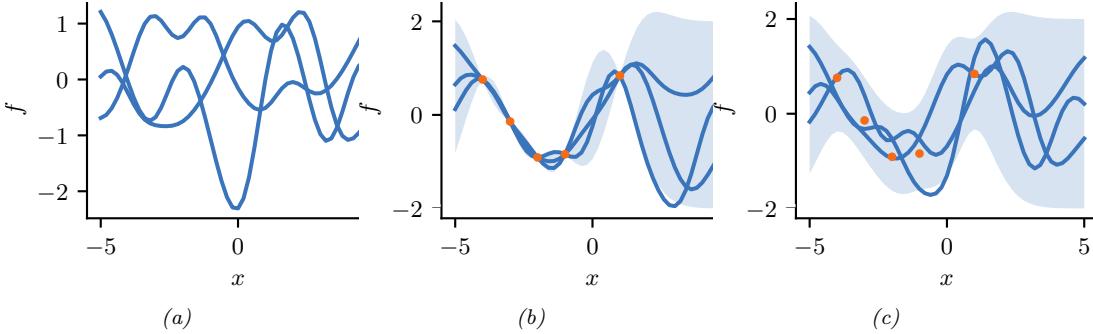


Figure 18.7: Left: some functions sampled from a GP prior with RBF kernel. Middle: some samples from a GP posterior, after conditioning on 5 noise-free observations. Right: some samples from a GP posterior, after conditioning on 5 noisy observations. The shaded area represents $\mathbb{E}[f(\mathbf{x})] \pm 2\sqrt{\text{Var}[f(\mathbf{x})]}$. Adapted from Figure 2.2 of [RW06]. Generated by [gpr_demo_noise_free.ipynb](#).

By definition of the GP, the joint distribution $p(\mathbf{f}_X, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$ has the following form

$$\begin{pmatrix} \mathbf{f}_X \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\top & \mathbf{K}_{*,*} \end{pmatrix} \right) \quad (18.44)$$

where $\boldsymbol{\mu}_X = (m(\mathbf{x}_1), \dots, m(\mathbf{x}_N))$, $\boldsymbol{\mu}_* = (m(\mathbf{x}_1^*), \dots, m(\mathbf{x}_{N_*}^*))$, $\mathbf{K}_{X,X} = \mathcal{K}(\mathbf{X}, \mathbf{X})$ is $N \times N$, $\mathbf{K}_{X,*} = \mathcal{K}(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$, and $\mathbf{K}_{*,*} = \mathcal{K}(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. See Figure 18.7 for a static illustration, and <http://www.infinitecuriosity.org/vizgp/> for an interactive visualization.

By the standard rules for conditioning Gaussians (Section 2.3.1.4), the posterior has the following form

$$p(\mathbf{f}_* | \mathbf{X}_*, \mathcal{D}) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.45)$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^\top \mathbf{K}_{X,X}^{-1} (\mathbf{f}_X - \boldsymbol{\mu}_X) \quad (18.46)$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\top \mathbf{K}_{X,X}^{-1} \mathbf{K}_{X,*} \quad (18.47)$$

This process is illustrated in Figure 18.7. On the left we show some samples from the prior, $p(f)$, where we use an RBF kernel (Section 18.2.1.1) and a zero mean function. On the right, we show samples from the posterior, $p(f|\mathcal{D})$. We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.

Note that the cost of the above method for sampling N_* points is $O(N_*^3)$. This can be reduced to $O(N_*)$ time using the methods in [Ple+18; Wil+20a].

18.3.2 Predictions using noisy observations

In Section 18.3.1, we showed how to do GP regression when the training data was noiseless. Now let us consider the case where what we observe is a noisy version of the underlying function, $y_n = f(\mathbf{x}_n) + \epsilon_n$, where $\epsilon_n \sim \mathcal{N}(0, \sigma_y^2)$. In this case, the model is not required to interpolate the data, but it must come “close” to the observed data. The covariance of the observed noisy responses is

$$\text{Cov}[y_i, y_j] = \text{Cov}[f_i, f_j] + \text{Cov}[\epsilon_i, \epsilon_j] = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) + \sigma_y^2 \delta_{ij} \quad (18.48)$$

where $\delta_{ij} = \mathbb{I}(i = j)$. In other words

$$\text{Cov}[\mathbf{y}|\mathbf{X}] = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I}_N \quad (18.49)$$

The joint density of the observed data and the latent, noise-free function on the test points is given by

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_X \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} & \mathbf{K}_{X,*} \\ \mathbf{K}_{X,*}^\top & \mathbf{K}_{*,*} \end{pmatrix}\right) \quad (18.50)$$

Hence the posterior predictive density at a set of test points \mathbf{X}_* is

$$p(\mathbf{f}_* | \mathcal{D}, \mathbf{X}_*) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.51)$$

$$\boldsymbol{\mu}_{*|X} = \boldsymbol{\mu}_* + \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) \quad (18.52)$$

$$\boldsymbol{\Sigma}_{*|X} = \mathbf{K}_{*,*} - \mathbf{K}_{X,*}^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{K}_{X,*} \quad (18.53)$$

In the case of a single test input, this simplifies as follows

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | m_* + \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} (\mathbf{y} - \boldsymbol{\mu}_X), k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_*) \quad (18.54)$$

where $\mathbf{k}_* = [\mathcal{K}(\mathbf{x}_*, \mathbf{x}_1), \dots, \mathcal{K}(\mathbf{x}_*, \mathbf{x}_N)]$ and $k_{**} = \mathcal{K}(\mathbf{x}_*, \mathbf{x}_*)$. If the mean function is zero, we can write the posterior mean as follows:

$$\mu_{*|X} = \mathbf{k}_*^\top \underbrace{\mathbf{K}_\sigma^{-1}}_{\boldsymbol{\alpha}} \mathbf{y} = \sum_{n=1}^N \mathcal{K}(\mathbf{x}_*, \mathbf{x}_n) \alpha_n \quad (18.55)$$

where

$$\mathbf{K}_\sigma = \mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I} \quad (18.56)$$

$$\boldsymbol{\alpha} = \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.57)$$

Fitting this model amounts to computing $\boldsymbol{\alpha}$ in Equation (18.57). This is usually done by computing the Cholesky decomposition of \mathbf{K}_σ , as described in Section 18.3.6. Once we have computed $\boldsymbol{\alpha}$, we can compute predictions for each test point in $O(N)$ time for the mean, and $O(N^2)$ time for the variance.

18.3.3 Weight space vs function space

In this section, we show how Bayesian linear regression is a special case of a GP.

Consider the linear regression model $y = f(\mathbf{x}) + \epsilon$, where $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$ and $\epsilon \sim \mathcal{N}(0, \sigma_y^2)$. If we use a Gaussian prior $p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \boldsymbol{\Sigma}_w)$, then the posterior is as follows (see Section 15.2.2 for the derivation):

$$p(\mathbf{w} | \mathcal{D}) = \mathcal{N}(\mathbf{w} | \frac{1}{\sigma_y^2} \mathbf{A}^{-1} \Phi^T \mathbf{y}, \mathbf{A}^{-1}) \quad (18.58)$$

where Φ is the $N \times D$ design matrix, and

$$\mathbf{A} = \sigma_y^{-2} \Phi^\top \Phi + \Sigma_w^{-1} \quad (18.59)$$

The posterior predictive distribution for $f_* = f(\mathbf{x}_*)$ is therefore

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \frac{1}{\sigma_y^2} \phi_*^\top \mathbf{A}^{-1} \Phi^\top \mathbf{y}, \phi_*^\top \mathbf{A}^{-1} \phi_*) \quad (18.60)$$

where $\phi_* = \phi(\mathbf{x}_*)$. This views the problem of inference and prediction in **weight space**.

We now show that this is equivalent to the predictions made by a GP using a kernel of the form $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \Sigma_w \phi(\mathbf{x}')$. To see this, let $\mathbf{K} = \Phi \Sigma_w \Phi^\top$, $\mathbf{k}_* = \Phi \Sigma_w \phi_*$, and $k_{**} = \phi_*^\top \Sigma_w \phi_*$. Using this notation, and the matrix inversion lemma, we can rewrite Equation (18.60) as follows

$$p(f_* | \mathcal{D}, \mathbf{x}_*) = \mathcal{N}(f_* | \boldsymbol{\mu}_{*|X}, \boldsymbol{\Sigma}_{*|X}) \quad (18.61)$$

$$\boldsymbol{\mu}_{*|X} = \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} = \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.62)$$

$$\boldsymbol{\Sigma}_{*|X} = \phi_*^\top \Sigma_w \phi_* - \phi_*^\top \Sigma_w \Phi^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \Phi \Sigma_w \phi_* = k_{**} - \mathbf{k}_*^\top (\mathbf{K}_{X,X} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{k}_* \quad (18.63)$$

which matches the results in Equation (18.54), assuming $m(\mathbf{x}) = 0$. A non-zero mean can be captured by adding a constant feature with value 1 to $\phi(\mathbf{x})$.

Thus we can derive a GP from Bayesian linear regression. Note, however, that linear regression assumes $\phi(\mathbf{x})$ is a finite length vector, whereas a GP allows us to work directly in terms of kernels, which may correspond to infinite length feature vectors (see Section 18.2.5). That is, a GP works in **function space**.

18.3.4 Semiparametric GPs

So far, we have mostly assumed the mean of the GP is 0, and have relied on its interpolation abilities to model the mean function. Sometimes it is useful to fit a global linear model for the mean, and use the GP to model the residual errors, as follows:

$$g(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\beta}^\top \phi(\mathbf{x}) \quad (18.64)$$

where $f(\mathbf{x}) \sim \text{GP}(0, \mathcal{K}(\mathbf{x}, \mathbf{x}'))$, and $\phi()$ are some fixed basis functions. This combines a parametric and a non-parametric model, and is known as a **semi-parametric model**.

If we assume $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{b}, \mathbf{B})$, we can integrate these parameters out to get a new GP [O'H78]:

$$g(\mathbf{x}) \sim \text{GP}(\phi(\mathbf{x})^\top \mathbf{b}, \mathcal{K}(\mathbf{x}, \mathbf{x}') + \phi(\mathbf{x})^\top \mathbf{B} \phi(\mathbf{x}')) \quad (18.65)$$

Let $\mathbf{H}_X = \phi(\mathbf{X})^\top$ be the $D \times N$ matrix of training examples, and $\mathbf{H}_* = \phi(\mathbf{X}_*)^\top$ be the $D \times N_*$ matrix of test examples. The corresponding predictive distribution for test inputs \mathbf{X}_* has the following form [RW06, p28]:

$$\mathbb{E}[g(\mathbf{X}_*) | \mathcal{D}] = \mathbf{H}_*^\top \bar{\boldsymbol{\beta}} + \mathbf{K}_{X,*}^\top \mathbf{K}_\sigma^{-1} (\mathbf{y} - \mathbf{H}_X^\top \bar{\boldsymbol{\beta}}) = \mathbb{E}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top \bar{\boldsymbol{\beta}} \quad (18.66)$$

$$\text{Cov}[g(\mathbf{X}_*) | \mathcal{D}] = \text{Cov}[f(\mathbf{X}_*) | \mathcal{D}] + \mathbf{R}^\top (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} \mathbf{R} \quad (18.67)$$

$$\bar{\boldsymbol{\beta}} = (\mathbf{B}^{-1} + \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^\top)^{-1} (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} + \mathbf{B}^{-1} \mathbf{b}) \quad (18.68)$$

$$\mathbf{R} = \mathbf{H}_* - \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{K}_{X,*} \quad (18.69)$$

These results can be interpreted as follows: the mean is the usual mean from the GP, plus a global offset from the linear model, using $\bar{\beta}$; and the covariance is the usual covariance from the GP, plus an additional positive term due to the uncertainty in β .

In the limit of an uninformative prior for the regression parameters, as $\mathbf{B} \rightarrow \infty \mathbf{I}$, this simplifies to

$$\mathbb{E}[g(\mathbf{X}_*)|\mathcal{D}] = \mathbb{E}[f(\mathbf{X}_*)|\mathcal{D}] + \mathbf{R}^T (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^T)^{-1} \mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{y} \quad (18.70)$$

$$\text{Cov}[g(\mathbf{X}_*)|\mathcal{D}] = \text{Cov}[f(\mathbf{X}_*)|\mathcal{D}] + \mathbf{R}^T (\mathbf{H}_X \mathbf{K}_\sigma^{-1} \mathbf{H}_X^T)^{-1} \mathbf{R} \quad (18.71)$$

18.3.5 Marginal likelihood

Most kernels have some free parameters. For example, the RBF-ARD kernel (Section 18.2.1.2) has the form

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp \left(-\frac{1}{2} \sum_{d=1}^D \frac{1}{\ell_d^2} (x_d - x'_d)^2 \right) = \prod_{d=1}^D \mathcal{K}_{\ell_d}(x_d, x'_d) \quad (18.72)$$

where each ℓ_d is a length scale for feature dimension d . Let these (and the observation noise variance σ_y^2 , if present) be denoted by θ . We can compute the likelihood of these parameters as follows:

$$p(\mathbf{y}|\mathbf{X}, \theta) = p(\mathcal{D}|\theta) = \int p(\mathbf{y}|\mathbf{f}_X, \theta) p(\mathbf{f}_X|\mathbf{X}, \theta) d\mathbf{f}_X \quad (18.73)$$

Since we are integrating out the function f , we often call θ hyperparameters, and the quantity $p(\mathcal{D}|\theta)$ the marginal likelihood.

Since f is a GP, we can compute the above integral using the marginal likelihood for the corresponding Gaussian. This gives

$$\log p(\mathcal{D}|\theta) = -\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu}_X)^T \mathbf{K}_\sigma^{-1} (\mathbf{y} - \boldsymbol{\mu}_X) - \frac{1}{2} \log |\mathbf{K}_\sigma| - \frac{N}{2} \log(2\pi) \quad (18.74)$$

The first term is the square of the Mahalanobis distance between the observations and the predicted values: better fits will have smaller distance. The second term is the log determinant of the covariance matrix, which measures model complexity: smoother functions will have smaller determinants, so $-\log |\mathbf{K}_\sigma|$ will be larger (less negative) for simpler functions. The marginal likelihood measures the tradeoff between fit and complexity.

In Section 18.6.1, we discuss how to learn the kernel parameters from data by maximizing the marginal likelihood wrt θ .

18.3.6 Computational and numerical issues

In this section, we discuss computational and numerical issues which arise when implementing the above equations. For notational simplicity, we assume the prior mean is zero, $m(\mathbf{x}) = 0$.

The posterior predictive mean is given by $\mu_* = \mathbf{k}_*^T \mathbf{K}_\sigma^{-1} \mathbf{y}$. For reasons of numerical stability, it is unwise to directly invert \mathbf{K}_σ . A more robust alternative is to compute a Cholesky decomposition, $\mathbf{K}_\sigma = \mathbf{L}\mathbf{L}^T$, which takes $O(N^3)$ time. Given this, we can compute

$$\mu_* = \mathbf{k}_*^T \mathbf{K}_\sigma^{-1} \mathbf{y} = \mathbf{k}_*^T \mathbf{L}^{-T} (\mathbf{L}^{-1} \mathbf{y}) = \mathbf{k}_*^T \boldsymbol{\alpha} \quad (18.75)$$

Here $\boldsymbol{\alpha} = \mathbf{L}^T \setminus (\mathbf{L} \setminus \mathbf{y})$, where we have used the backslash operator to represent backsubstitution.

We can compute the variance in $O(N^2)$ time for each test case using

$$\sigma_*^2 = k_{**} - \mathbf{k}_*^T \mathbf{L}^{-T} \mathbf{L}^{-1} \mathbf{k}_* = k_{**} - \mathbf{v}^T \mathbf{v} \quad (18.76)$$

where $\mathbf{v} = \mathbf{L} \setminus \mathbf{k}_*$.

Finally, the log marginal likelihood (needed for kernel learning, Section 18.6) can be computed using

$$\log p(\mathbf{y} | \mathbf{X}) = -\frac{1}{2} \mathbf{y}^T \boldsymbol{\alpha} - \sum_{n=1}^N \log L_{nn} - \frac{N}{2} \log(2\pi) \quad (18.77)$$

We see that overall cost is dominated by $O(N^3)$. We discuss faster, but approximate, methods in Section 18.5.

18.3.7 Kernel ridge regression

The term **ridge regression** refers to linear regression with an ℓ_2 penalty on the regression weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + \lambda \|\mathbf{w}\|_2^2 \quad (18.78)$$

where $f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \mathbf{x}$. The solution for this is

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^T + \lambda \mathbf{I} \right)^{-1} \left(\sum_{n=1}^N \mathbf{x}_n y_n \right) \quad (18.79)$$

In this section, we consider a function space version of this:

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \lambda \|f\|^2 \quad (18.80)$$

For this to make sense, we have to define the function space \mathcal{F} and the norm $\|f\|$. If we use a function space derived from a positive definite kernel function \mathcal{K} , the resulting method is called **kernel ridge regression** (KRR). We will see that the resulting estimate $f^*(\mathbf{x}_*)$ is equivalent to the posterior mean of a GP. We give the details below.

18.3.7.1 Reproducing kernel Hilbert spaces

In this section, we briefly introduce the relevant mathematical ‘‘machinery’’ needed to explain KRR.

Let $\mathcal{F} = \{f : \mathcal{X} \rightarrow \mathbb{R}\}$ be a space of real-valued functions. Elements of this space (i.e., functions) can be added and scalar multiplied as if they were vectors. That is, if $f \in \mathcal{F}$ and $g \in \mathcal{F}$, then $\alpha f + \beta g \in \mathcal{F}$ for $\alpha, \beta \in \mathbb{R}$. We can also define an **inner product** for \mathcal{F} , which is a mapping $\langle f, g \rangle \in \mathbb{R}$

which satisfies the following:

$$\langle \alpha f_1 + \beta f_2, g \rangle = \alpha \langle f_1, g \rangle + \beta \langle f_2, g \rangle \quad (18.81)$$

$$\langle f, g \rangle = \langle g, f \rangle \quad (18.82)$$

$$\langle f, f \rangle \geq 0 \quad (18.83)$$

$$\langle f, f \rangle = 0 \text{ iff } f(x) = 0 \text{ for all } x \in \mathcal{X} \quad (18.84)$$

We define the norm of a function using

$$\|f\| \triangleq \sqrt{\langle f, f \rangle} \quad (18.85)$$

A function space \mathcal{H} with an inner product operator is called a **Hilbert space**. (We also require that the function space be complete, which means that every Cauchy sequence of functions $f_i \in \mathcal{H}$ has a limit that is also in \mathcal{H} .)

The most common Hilbert space is the space known as L^2 . To define this, we need to specify a **measure** μ on the input space \mathcal{X} ; this is a function that assigns any (suitable) subset A of \mathcal{X} to a positive number, such as its volume. This can be defined in terms of the density function $w : \mathcal{X} \rightarrow \mathbb{R}$, as follows:

$$\mu(A) = \int_A w(x) dx \quad (18.86)$$

Thus we have $\mu(dx) = w(x)dx$. We can now define $L^2(\mathcal{X}, \mu)$ to be the space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$ that satisfy

$$\int_{\mathcal{X}} f(x)^2 w(x) dx < \infty \quad (18.87)$$

This is known as the set of **square-integrable functions**. This space has an inner product defined by

$$\langle f, g \rangle = \int_{\mathcal{X}} f(x)g(x)w(x) dx \quad (18.88)$$

We define a **Reproducing Kernel Hilbert Space** or **RKHS** as follows. Let \mathcal{H} be a Hilbert space of functions $f : \mathcal{X} \rightarrow \mathbb{R}$. We say that \mathcal{H} is an RKHS endowed with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ if there exists a (symmetric) **kernel function** $\mathcal{K} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ with the following properties:

- For every $\mathbf{x} \in \mathcal{X}$, $\mathcal{K}(\mathbf{x}, \cdot) \in \mathcal{H}$.
- \mathcal{K} satisfies the **reproducing property**:

$$\langle f(\cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = f(\mathbf{x}') \quad (18.89)$$

The reason for the term “reproducing property” is as follows. Let $f(\cdot) = \mathcal{K}(\mathbf{x}, \cdot)$. Then we have that

$$\langle \mathcal{K}(\mathbf{x}, \cdot), \mathcal{K}(\cdot, \mathbf{x}') \rangle = \mathcal{K}(\mathbf{x}, \mathbf{x}') \quad (18.90)$$

18.3.7.2 Complexity of a function in an RKHS

The main utility of RKHS from the point of view of machine learning is that it allows us to define a notion of a function's "smoothness" or "complexity" in terms of its norm, as we now discuss.

Suppose we have a positive definite kernel function \mathcal{K} . From Mercer's theorem we have $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{\infty} \lambda_i \phi_i(\mathbf{x}) \phi_i(\mathbf{x}')$. Now consider a Hilbert space \mathcal{H} defined by functions of the form $f(\mathbf{x}) = \sum_{i=1}^{\infty} f_i \phi_i(\mathbf{x})$, with $\sum_{i=1}^{\infty} f_i^2 / \lambda_i < \infty$. The inner product of two functions in this space is

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i g_i}{\lambda_i} \quad (18.91)$$

Hence the (squared) norm is given by

$$\|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i^2}{\lambda_i} \quad (18.92)$$

This is analogous to the quadratic form $\mathbf{f}^T \mathbf{K}^{-1} \mathbf{f}$ which occurs in some GP objectives (see Equation (18.101)). Thus the smoothness of the function is controlled by the properties of the corresponding kernel.

18.3.7.3 Representer theorem

In this section, we consider the problem of (regularized) empirical risk minimization in function space. In particular, consider the following problem:

$$f^* = \underset{f \in \mathcal{H}_{\mathcal{K}}}{\operatorname{argmin}} \sum_{n=1}^N \ell(y_n, f(\mathbf{x}_n)) + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2 \quad (18.93)$$

where $\mathcal{H}_{\mathcal{K}}$ is an RKHS with kernel \mathcal{K} and $\ell(y, \hat{y}) \in \mathbb{R}$ is a loss function. Then one can show [KW70; SHS01] the following result:

$$f^*(x) = \sum_{n=1}^N \alpha_n \mathcal{K}(x, \mathbf{x}_n) \quad (18.94)$$

where $\alpha_n \in \mathbb{R}$ are some coefficients that depend on the training data. This is called the **representer theorem**.

Now consider the special case where the loss function is squared loss, and $\lambda = \sigma_y^2$. We want to minimize

$$\mathcal{L}(f) = \frac{1}{2\sigma_y^2} \sum_{n=1}^N (y_n - f(\mathbf{x}_n))^2 + \frac{1}{2} \|f\|_{\mathcal{H}}^2 \quad (18.95)$$

Substituting in Equation (18.94), and using the fact that $\langle \mathcal{K}(\cdot, \mathbf{x}_i), \mathcal{K}(\cdot, \mathbf{x}_j) \rangle = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$, we obtain

$$\mathcal{L}(f) = \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \|\mathbf{y} - \mathbf{K} \boldsymbol{\alpha}\|^2 \quad (18.96)$$

$$= \frac{1}{2} \boldsymbol{\alpha}^T (\mathbf{K} + \frac{1}{\sigma_y^2} \mathbf{K}^2) \boldsymbol{\alpha} - \frac{1}{\sigma_y^2} \mathbf{y}^T \mathbf{K} \boldsymbol{\alpha} + \frac{1}{2\sigma_y^2} \mathbf{y}^T \mathbf{y} \quad (18.97)$$

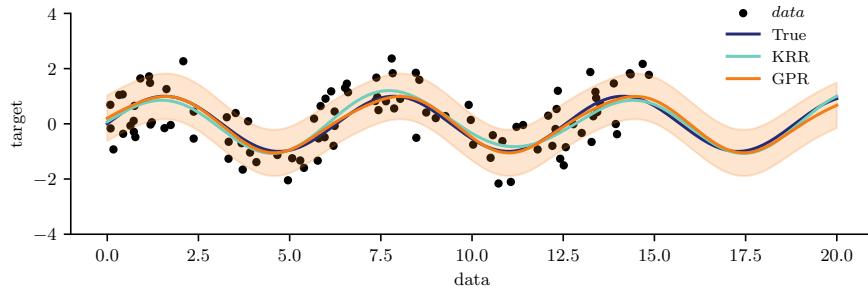


Figure 18.8: Kernel ridge regression (KRR) compared to Gaussian process regression (GPR) using the same kernel. Generated by [krr_vs_gpr.ipynb](#).

Minimizing this wrt α gives $\hat{\alpha} = (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y}$, which is the same as Equation (18.57). Furthermore, the prediction for a test point is

$$\hat{f}(\mathbf{x}_*) = \mathbf{k}_*^\top \alpha = \mathbf{k}_*^\top (\mathbf{K} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y} \quad (18.98)$$

This is known as **kernel ridge regression** [Vov13]. We see that the result matches the posterior predictive mean of a GP in Equation (18.55).

18.3.7.4 Example of KRR vs GPR

In this section, we compare KRR with GP regression on a simple 1d problem. Since the underlying function is believed to be periodic, we use the periodic kernel from Equation (18.18). To capture the fact that the observations are noisy, we add to this a **white noise kernel**

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sigma_y^2 \delta(\mathbf{x} - \mathbf{x}') \quad (18.99)$$

as in Equation (18.48). Thus there are 3 GP hyper-parameters: the kernel length scale ℓ , the kernel periodicity p , and the noise level σ_y^2 . We can optimize these by maximizing the marginal likelihood using gradient descent (see Section 18.6.1). For KRR, we also have 3 hyperparameters (ℓ , p , and $\lambda = \sigma_y^2$); we optimize these using grid search combined with cross validation (which in general is slower than gradient based optimization). The resulting model fits are shown in Figure 18.8, and are very similar, as is to be expected.

18.4 GPs with non-Gaussian likelihoods

So far, we have focused on GPs for regression using Gaussian likelihoods. In this case, the posterior is also a GP, and all computation can be performed analytically. However, if the likelihood is non-Gaussian, we can no longer compute the posterior exactly. We can create variety of different “classical” models by changing the form of the likelihood, as we show in Table 18.1. In the sections below, we briefly discuss some approximate inference methods. (For more details, see e.g., [WSS21].)

Model	Likelihood	Section
Regression	$\mathcal{N}(f_i, \sigma_y^2)$	Section 18.3.2
Robust regression	$T_\nu(f_i, \sigma_y^2)$	Section 18.4.4
Binary classification	$\text{Ber}(\sigma(f_i))$	Section 18.4.1
Multiclass classification	$\text{Cat}(\text{softmax}(\mathbf{f}_i))$	Section 18.4.2
Poisson regression	$\text{Poi}(\exp(f_i))$	Section 18.4.3

Table 18.1: Summary of GP models with a variety of likelihoods.

$$\begin{array}{c|c|c} \log p(y_i|f_i) & \frac{\partial}{\partial f_i} \log p(y_i|f_i) & \frac{\partial^2}{\partial f_i^2} \log p(y_i|f_i) \\ \hline \log \sigma(y_i f_i) & t_i - \pi_i & -\pi_i(1 - \pi_i) \\ \log \Phi(y_i f_i) & \frac{y_i \phi(f_i)}{\Phi(y_i f_i)} & -\frac{\phi_i^2}{\Phi(y_i f_i)^2} - \frac{y_i f_i \phi(f_i)}{\Phi(y_i f_i)} \end{array}$$

Table 18.2: Likelihood, gradient, and Hessian for binary logistic/probit GP regression. We assume $y_i \in \{-1, +1\}$ and define $t_i = (y_i + 1)/2 \in \{0, 1\}$ and $\pi_i = \sigma(f_i)$ for logistic regression, and $\pi_i = \Phi(f_i)$ for probit regression. Also, ϕ and Φ are the pdf and cdf of $\mathcal{N}(0, 1)$. From [RW06, p43].

18.4.1 Binary classification

In this section, we consider binary classification using GPs. If we use the sigmoid link function, we have $p(y_n = 1|\mathbf{x}_n) = \sigma(y_n f(\mathbf{x}_n))$. If we assume $y_n \in \{-1, +1\}$, then we have $p(y_n|\mathbf{x}_n) = \sigma(y_n f_n)$, since $\sigma(-z) = 1 - \sigma(z)$. If we use the probit link, we have $p(y_n = 1|\mathbf{x}_n) = \Phi(y_n f(\mathbf{x}_n))$, where $\Phi(z)$ is the cdf of the standard normal. More generally, let $p(y_n|\mathbf{x}_n) = \text{Ber}(y_n|\varphi(f_n))$. The overall log joint has the form

$$\mathcal{L}(\mathbf{f}_X) = \log p(\mathbf{y}|\mathbf{f}_X) + \log p(\mathbf{f}_X|\mathbf{X}) \quad (18.100)$$

$$= \log p(\mathbf{y}|\mathbf{f}_X) - \frac{1}{2} \mathbf{f}_X^\top \mathbf{K}_{X,X}^{-1} \mathbf{f}_X - \frac{1}{2} \log |\mathbf{K}_{X,X}| - \frac{N}{2} \log 2\pi \quad (18.101)$$

The simplest approach to approximate inference is to use a Laplace approximation (Section 7.4.3). The gradient and Hessian of the log joint are given by

$$\nabla \mathcal{L} = \nabla \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} \mathbf{f}_X \quad (18.102)$$

$$\nabla^2 \mathcal{L} = \nabla^2 \log p(\mathbf{y}|\mathbf{f}_X) - \mathbf{K}_{X,X}^{-1} = -\boldsymbol{\Lambda} - \mathbf{K}_{X,X}^{-1} \quad (18.103)$$

where $\boldsymbol{\Lambda} \triangleq -\nabla^2 \log p(\mathbf{y}|\mathbf{f}_X)$ is a diagonal matrix, since the likelihood factorizes across examples. Expressions for the gradient and Hessian of the log likelihood for the logit and probit case are shown in Table 18.2. At convergence, the Laplace approximation of the posterior takes the following form:

$$p(\mathbf{f}_X|\mathcal{D}) \approx q(\mathbf{f}_X) = \mathcal{N}(\hat{\mathbf{f}}, (\mathbf{K}_{X,X}^{-1} + \boldsymbol{\Lambda})^{-1}) \quad (18.104)$$

where $\hat{\mathbf{f}}$ is the MAP estimate. See [RW06, Sec 3.4] for further details.

For improved accuracy, we can use variational inference, in which we assume $q(\mathbf{f}_X) = \mathcal{N}(\mathbf{f}_X|\mathbf{m}, \mathbf{S})$; we then optimize \mathbf{m} and \mathbf{S} using (stochastic) gradient descent, rather than assuming \mathbf{S} is the Hessian at the mode. See Section 18.5.4 for the details.

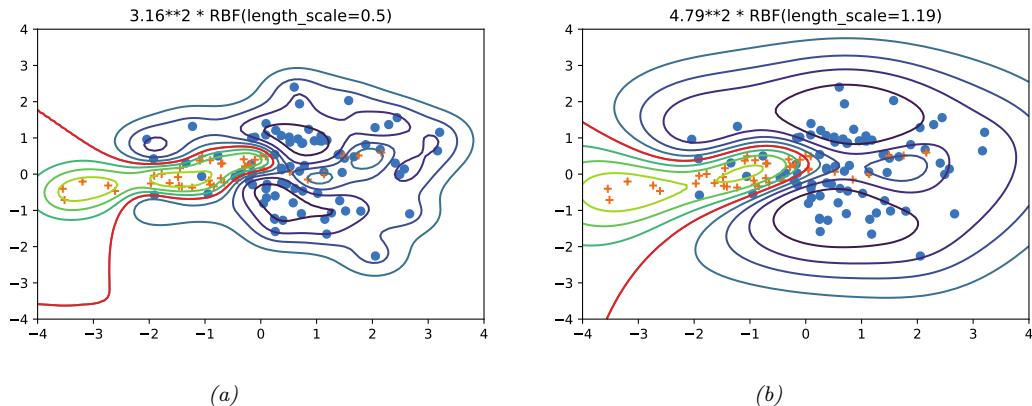


Figure 18.9: Contours of the posterior predictive probability for a binary classifier generated by a GP with an SE kernel. (a) Manual kernel parameters: short length scale, $\ell = 0.5$, variance $3.16^2 \approx 9.98$. (b) Learned kernel parameters: long length scale, $\ell = 1.19$, variance $4.79^2 \approx 22.9$. Generated by [gpc_demo_2d.ipynb](#).

Once we have a Gaussian posterior $q(\mathbf{f}_X | \mathcal{D})$, we can then use standard GP prediction to compute $q(f_* | \mathbf{x}_*, \mathcal{D})$. Finally, we can approximate the posterior predictive distribution over binary labels using

$$\pi_* = p(y_* = 1 | \boldsymbol{x}_*, \mathcal{D}) = \int p(y_* = 1 | f_*) q(f_* | \boldsymbol{x}_*, \mathcal{D}) df_* \quad (18.105)$$

This 1d integral can be computed using the probit approximation from Section 15.3.6. In this case we have $\pi_* \approx \sigma(\kappa(v)\mathbb{E}[f_*])$, where $v = \mathbb{V}[f_*]$ and $\kappa^2(v) = (1 + \pi v / 8)^{-1}$.

In Figure 18.9, we show a synthetic binary classification problem in 2d. We use an SE kernel. On the left, we show predictions using hyper-parameters set by hand; we use a short length scale, hence the very sharp turns in the decision boundary. On the right, we show the predictions using the learned hyper-parameters; the model favors more parsimonious explanation of the data.

18.4.2 Multiclass classification

The multi-class case is somewhat harder, since the function now needs to return a vector of C logits to get $p(y_n|\mathbf{x}_n) = \text{Cat}(y_n|\text{softmax}(\mathbf{f}_n))$, where $\mathbf{f}_n = (f_n^1, \dots, f_n^C)$. It is standard to assume that $f^c \sim \text{GP}(0, \mathcal{K}_c)$. Thus we have one latent function per class, which are a priori independent, and which may use different kernels.

We can derive a Laplace approximation for this model as discussed in [RW06, Sec 3.5]. Alternatively, we can use a variational approach, using the local variational bound to the multinomial softmax in [Cha12]. An alternative variational method, based on data augmentation with auxiliary variables, is described in [Wen+19b; Liu+19a; GFWO20].

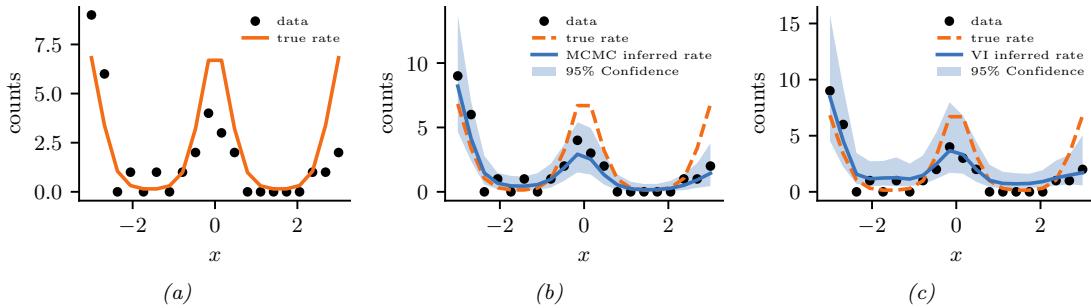


Figure 18.10: Poisson regression with a GP. (a) Observed data (black dots) and true log rate function (yellow line). (b) Posterior predictive distribution (shading shows 1 and 2 σ bands) from MCMC. (c) Posterior predictive distribution from SVI. Generated by [gp_poisson_1d.ipynb](#).

18.4.3 GPs for Poisson regression (Cox process)

In this section, we illustrate Poisson regression where the underlying log rate function is modeled by a GP. This is known as a **Cox process**. We can perform approximate posterior inference in this model using Laplace, MCMC, or SVI (stochastic variational inference). In Figure 18.10 we give a 1d example, where we use a Matérn $\frac{5}{2}$ kernel. We apply MCMC and SVI. In the VI case, we additionally have to specify the form of the posterior; we use a Gaussian approximation for the variational GP posterior $p(\mathbf{f}|\mathbf{X}, \mathbf{y})$, and a point estimate for the kernel parameters.

An interesting application of this is to spatial **disease mapping**. For example, [VPV10] discuss the problem of modeling the relative risk of heart attack in different regions in Finland. The data consists of the heart attacks in Finland from 1996–2000 aggregated into $20\text{km} \times 20\text{km}$ lattice cells. The likelihood has the following form: $y_n \sim \text{Poi}(e_n r_n)$, where e_n is the known expected number of deaths (related to the population of cell n and the overall death rate), and r_n is the **relative risk** of cell n which we want to infer. Since the data counts are small, we regularize the problem by sharing information with spatial neighbors. Hence we assume $f \triangleq \log(r) \sim \text{GP}(0, \mathcal{K})$. We use a Matérn kernel (Section 18.2.1.3) with $\nu = 3/2$, and a length scale and magnitude that are estimated from data.

Figure 18.11 gives an example of this method in action (using Laplace approximation). On the left we plot the posterior mean relative risk (RR), and on the right, the posterior variance. We see that the RR is higher in eastern Finland, which is consistent with other studies. We also see that the variance in the north is higher, since there are fewer people living there.

18.4.4 Other likelihoods

Many other likelihoods are possible. For example, [VJV09] uses a Student t likelihood in order to perform robust regression. A general method for performing approximate variational inference in GPs with such non-conjugate likelihoods is discussed in [WSS21].

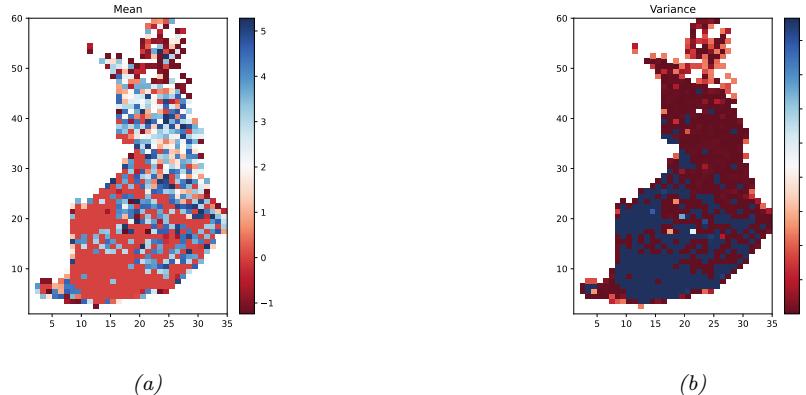


Figure 18.11: We show the relative risk of heart disease in Finland using a Poisson GP fit to 911 data points. Left: posterior mean. Right: posterior variance. Generated by [gp_spatial_demo.ipynb](#).

Method	Cost	Section
Cholesky	$O(N^3)$	Section 18.3.6
Conj. Grad.	$O(CN^2)$	Section 18.5.5
Inducing	$O(NM^2 + M^3 + DNM)$	Section 18.5.3
Variational	$O(NM^2 + M^3 + DNM)$	Section 18.5.4
SVGP	$O(BM^2 + M^3 + DNM)$	Section 18.5.4.3
KISS-GP	$O(CN + CDM^D \log M)$	Section 18.5.5.3
SKIP	$O(DLN + DLM \log M + L^3 N \log D + CL^2 N)$	Section 18.5.5.3

Table 18.3: Summary of time to compute the log marginal likelihood of a GP regression model. Notation: N is number of training examples, M is number of inducing points, B is size of minibatch, D is dimensionality of input vectors (assuming $\mathcal{X} = \mathbb{R}^D$), C is number of conjugate gradient iterations, and L is number of Lanczos iterations. Based on Table 2 of [Gar+18a].

18.5 Scaling GP inference to large datasets

In Section 18.3.6, we saw that the best way to perform GP inference and training is to compute a Cholesky decomposition of the $N \times N$ Gram matrix. Unfortunately, this takes $O(N^3)$ time. In this section, we discuss methods to scale up GPs to handle large N . See Table 18.3 for a summary, and [Liu+20c] for more details.¹

18.5.1 Subset of data

The simplest approach to speeding up GP inference is to throw away some of the data. Suppose we keep a subset of M examples. In this case, exact inference will take $O(M^3)$ time. This is called the

1. We focus on efficient methods for evaluating the marginal likelihood and the posterior predictive distribution. For an efficient method for sampling a function from the posterior, see [Wil+20a].

subset-of-data approach.

The key question is: how should we choose the subset? The simplest approach is to pick random examples (this method was recently analyzed in [HIY19]). However, intuitively it makes more sense to try to pick a subset that in some sense “covers” the original data, so it contains approximately the same information (up to some tolerance) without the redundancy. Clustering algorithms are one heuristic approach, but we can also use coresets methods, which can provably find such an information-preserving subset (see e.g., [Hug+19] for an application of this idea to GPs).

18.5.1.1 Informative vector machine

Clustering and coresets methods are unsupervised, in that they only look at the features \mathbf{x}_i and not the labels y_i , which can be suboptimal. The **informative vector machine** [HLS03] uses a greedy strategy to iteratively add the labeled example (\mathbf{x}_j, y_j) that maximally reduces the entropy of the function’s posterior, $\Delta_j = \mathbb{H}(p(f_j)) - \mathbb{H}(p^{\text{new}}(f_j))$, where $p^{\text{new}}(f_j)$ is the posterior of f at \mathbf{x}_j after conditioning on y_j . (This is very similar to active learning.) To compute Δ_j , let $p(f_j) = \mathcal{N}(\mu_j, v_j)$, and $p(f_j|y_j) \propto p(f_j)\mathcal{N}(y_j|f_j, \sigma^2) = \mathcal{N}(f_j|\mu_j^{\text{new}}, v_j^{\text{new}})$, where $(v_j^{\text{new}})^{-1} = v_j^{-1} + \sigma^{-2}$. Since $\mathbb{H}(\mathcal{N}(\mu, v)) = \log(2\pi ev)/2$, we have $\Delta_j = 0.5 \log(1 + v_j/\sigma^2)$. Since this is a monotonic function of v_j , we can maximize it by choosing the site with the largest variance. (In fact, entropy is a submodular function, so we can use submodular optimization algorithms to improve on the IVM, as shown in [Kra+08].)

18.5.1.2 Discussion

The main problem with the subset of data approach is that it ignores some of the data, which can reduce predictive accuracy and increase uncertainty about the true function. Fortunately there are other scalable methods that avoid this problem, essentially by approximately representing (or compressing) the training data, as we discuss below.

18.5.2 Nyström approximation

Suppose we had a rank M approximation to the $N \times N$ matrix gram matrix of the following form:

$$\mathbf{K}_{X,X} \approx \mathbf{U}\Lambda\mathbf{U}^\top \tag{18.106}$$

where Λ is a diagonal matrix of the M leading eigenvalues, and \mathbf{U} is the matrix of the corresponding M eigenvectors, each of size N . In this case, we can use the matrix inversion lemma to write

$$\mathbf{K}_\sigma^{-1} = (\mathbf{K}_{X,X} + \sigma^2 \mathbf{I}_N)^{-1} \approx \sigma^{-2} \mathbf{I}_N + \sigma^{-2} \mathbf{U}(\sigma^2 \Lambda^{-1} + \mathbf{U}^\top \mathbf{U})^{-1} \mathbf{U}^\top \tag{18.107}$$

which takes $O(NM^2)$ time. Similarly, one can show (using the Sylvester determinant lemma) that

$$|\mathbf{K}_\sigma| \approx |\Lambda| |\sigma^2 \Lambda^{-1} + \mathbf{U}^\top \mathbf{U}| \tag{18.108}$$

which also takes $O(NM^2)$ time.

Unfortunately, directly computing such an eigendecomposition takes $O(N^3)$ time, which does not help. However, suppose we pick a subset Z of $M < N$ points. We can partition the Gram matrix as

follows (where we assume the chosen points come first, and then the remaining points):

$$\mathbf{K}_{X,X} = \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,X-Z} \\ \mathbf{K}_{X-Z,Z} & \mathbf{K}_{X-Z,X-Z} \end{pmatrix} \triangleq \begin{pmatrix} \mathbf{K}_{Z,Z} & \mathbf{K}_{Z,\tilde{X}} \\ \mathbf{K}_{\tilde{X},Z} & \mathbf{K}_{\tilde{X},\tilde{X}} \end{pmatrix} \quad (18.109)$$

where $\tilde{X} = X - Z$. We now compute an eigendecomposition of $\mathbf{K}_{Z,Z}$ to get the eigenvalues $\{\lambda_i\}_{i=1}^M$ and eigenvectors $\{\mathbf{u}_i\}_{i=1}^M$. We now use these to approximate the full matrix as shown below, where the scaling constants are chosen so that $\|\tilde{\mathbf{u}}_i\| \approx 1$:

$$\tilde{\lambda}_i \triangleq \frac{N}{M} \lambda_i \quad (18.110)$$

$$\tilde{\mathbf{u}} \triangleq \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{\tilde{X},Z} \mathbf{u}_i \quad (18.111)$$

$$\mathbf{K}_{X,X} \approx \sum_{i=1}^M \tilde{\lambda}_i \tilde{\mathbf{u}}_i \tilde{\mathbf{u}}_i^\top \quad (18.112)$$

$$= \sum_{i=1}^M \frac{N}{M} \lambda_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{K}_{\tilde{X},Z} \mathbf{u}_i \sqrt{\frac{M}{N}} \frac{1}{\lambda_i} \mathbf{u}_i^\top \mathbf{K}_{\tilde{X},Z}^\top \quad (18.113)$$

$$= \mathbf{K}_{\tilde{X},Z} \left(\sum_{i=1}^M \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^\top \right) \mathbf{K}_{\tilde{X},Z} \quad (18.114)$$

$$= \mathbf{K}_{\tilde{X},Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{\tilde{X},Z}^\top \quad (18.115)$$

This is known as the **Nyström approximation** [WS01]. If we define

$$\mathbf{Q}_{A,B} \triangleq \mathbf{K}_{A,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,B} \quad (18.116)$$

then we can write the approximate Gram matrix as $\mathbf{Q}_{X,X}$. We can then replace \mathbf{K}_σ with $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$. Computing the eigendecomposition takes $O(M^3)$ time, and computing $\hat{\mathbf{Q}}_{X,X}^{-1}$ takes $O(NM^2)$ time. Thus complexity is now linear in N instead of cubic.

If we are approximating *only* $\hat{\mathbf{K}}_{X,X}$ in $\mu_{*|X}$ in Equation (18.52) and $\Sigma_{*|X}$ in Equation (18.53), then this is inconsistent with the other un-approximated kernel function evaluations in these formulae, and can result in the predictive variance being negative. One solution to this is to use the same \mathbf{Q} approximation for all terms.

18.5.3 Inducing point methods

In this section, we discuss an approximation method based on **inducing points**, also called **pseudoinputs**, which are like a learned summary of the training data that we can condition on, rather than conditioning on all of it.

Let \mathbf{X} be the observed inputs, and $\mathbf{f}_X = f(\mathbf{X})$ be the unknown vector of function values (for which we have noisy observations \mathbf{y}). Let \mathbf{f}_* be the unknown function values at one or more test points \mathbf{X}_* . Finally, let us assume we have M additional inputs, \mathbf{Z} , with unknown function values \mathbf{f}_Z (often denoted by \mathbf{u}). The exact joint prior has the form

$$p(\mathbf{f}_X, \mathbf{f}_*) = \int p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) d\mathbf{f}_Z = \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) d\mathbf{f}_Z = \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_{X,X} & \mathbf{K}_{X,*} \\ \mathbf{K}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}\right) \quad (18.117)$$



Figure 18.12: Illustration of the graphical model for a GP on n observations, $\mathbf{f}_{1:n}$, and one test case, f_* , with inducing variables \mathbf{u} . The thick lines indicate that all variables are fully interconnected. The observations y_i (not shown) are locally connected to each f_i . (a) no approximations are made. (b) we assume f_* is conditionally independent of \mathbf{f}_X given \mathbf{u} . From Figure 1 of [QCR05]. Used with kind permission of Joaquin Quiñonero Candela.

(We write $p(\mathbf{f}_X, \mathbf{f}_*)$ instead of $p(\mathbf{f}_X, \mathbf{f}_* | \mathbf{X}, \mathbf{X}_*)$, since the inputs can be thought of as just indices into the random function f .)

We will choose \mathbf{f}_Z in such a way that it acts as a sufficient statistic for the data, so that we can predict \mathbf{f}_* just using \mathbf{f}_Z instead of \mathbf{f}_X , i.e., we assume $\mathbf{f}_* \perp \mathbf{f}_X | \mathbf{f}_Z$. Thus we approximate the prior as follows:

$$p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) = p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z) \approx p(\mathbf{f}_* | \mathbf{f}_Z)p(\mathbf{f}_X | \mathbf{f}_Z)p(\mathbf{f}_Z) \quad (18.118)$$

See Figure 18.12 for an illustration of this assumption, and Section 18.5.3.4 for details on how to choose the inducing set \mathbf{Z} . (Note that this method is often called a “sparse GP”, because it makes predictions for \mathbf{f}_* using a subset of the training data, namely \mathbf{f}_Z , instead of all of it, \mathbf{f}_X .)

From this, we can derive the following train and test conditionals

$$p(\mathbf{f}_X | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{f}_Z, \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.119)$$

$$p(\mathbf{f}_* | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_* | \mathbf{K}_{*,Z}\mathbf{K}_{Z,Z}^{-1}\mathbf{f}_Z, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}) \quad (18.120)$$

The above equations can be seen as exact inference on noise-free observations \mathbf{f}_Z . To gain computational speedups, we will make further approximations to the terms $\tilde{\mathbf{Q}}_{X,X} = \mathbf{K}_{X,X} - \mathbf{Q}_{X,X}$ and $\tilde{\mathbf{Q}}_{*,*} = \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$, as we discuss below. We can then derive the approximate prior $q(\mathbf{f}_X, \mathbf{f}_*) = \int q(\mathbf{f}_X | \mathbf{f}_Z)q(\mathbf{f}_* | \mathbf{f}_Z)p(\mathbf{f}_Z)d\mathbf{f}_Z$, which we then condition on the observations in the usual way.

All of the approximations we discuss below result in an initial training cost of $O(M^3 + NM^2)$, and then take $O(M)$ time for the predictive mean for each test case, and $O(M^2)$ time for the predictive variance. (Compare this to $O(N^3)$ training time and $O(N)$ and $O(N^2)$ testing time for exact inference.)

18.5.3.1 SOR/DIC

Suppose we assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$ and $\tilde{\mathbf{Q}}_{*,*} = \mathbf{0}$, so the conditionals are deterministic. This is called the **deterministic inducing conditional (DIC)** approximation [QCR05], or the **subset of regressors (SOR)** approximation [Sil85; SB01]. The corresponding joint prior has the form

$$q_{\text{SOR}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{Q}_{*,*} \end{pmatrix}) \quad (18.121)$$

Let us define $\hat{\mathbf{Q}}_{X,X} = \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N$, and $\Sigma = (\sigma^{-2} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} + \mathbf{K}_{Z,Z})^{-1}$. Then the predictive distribution is

$$q_{\text{SOR}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{Q}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.122)$$

$$= \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,*} \Sigma \mathbf{K}_{Z,*}) \quad (18.123)$$

This is equivalent to the usual one for GPs except we have replaced $\mathbf{K}_{X,X}$ by $\hat{\mathbf{Q}}_{X,X}$. This is equivalent to performing GP inference with the following kernel function

$$\mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) = \mathcal{K}(\mathbf{x}_i, \mathbf{Z}) \mathbf{K}_{Z,Z}^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_j) \quad (18.124)$$

The kernel matrix has rank M , so the GP is degenerate. Furthermore, the kernel will be near 0 when \mathbf{x}_i or \mathbf{x}_j is far from one of the chosen points \mathbf{Z} , which can result in an underestimate of the predictive variance.

18.5.3.2 DTC

One way to overcome the overconfidence of DIC is to only assume $\tilde{\mathbf{Q}}_{X,X} = \mathbf{0}$, but let $\tilde{\mathbf{Q}}_{*,*} = \mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ be exact. This is called the **deterministic training conditional** or **DTC** method [SWL03].

The corresponding joint prior has the form

$$q_{\text{dtc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.125)$$

Hence the predictive distribution becomes

$$q_{\text{dtc}}(\mathbf{f}_* | \mathbf{y}) = \mathcal{N}(\mathbf{f}_* | \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,X} \hat{\mathbf{Q}}_{X,X}^{-1} \mathbf{Q}_{X,*}) \quad (18.126)$$

$$= \mathcal{N}(\mathbf{f}_* | \sigma^{-2} \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,X} \mathbf{y}, \mathbf{K}_{*,*} - \mathbf{Q}_{*,*} + \mathbf{K}_{*,Z} \Sigma \mathbf{K}_{Z,*}) \quad (18.127)$$

The predictive mean is the same as in SOR, but the variance is larger (since $\mathbf{K}_{*,*} - \mathbf{Q}_{*,*}$ is positive definite) due to the uncertainty of \mathbf{f}_* given \mathbf{f}_Z .

18.5.3.3 FITC

A widely used approximation assumes $q(\mathbf{f}_X | \mathbf{f}_Z)$ is fully factorized, i.e,

$$q(\mathbf{f}_X | \mathbf{f}_Z) = \prod_{n=1}^N p(f_n | \mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_X | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{f}_Z, \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X})) \quad (18.128)$$

This is called the **fully independent training conditional** or **FITC** assumption, and was first proposed in [SG06a]. This throws away less uncertainty than the SOR and DTC methods, since it does not make any deterministic assumptions about the relationship between \mathbf{f}_X and \mathbf{f}_Z .

The joint prior has the form

$$q_{\text{fitc}}(\mathbf{f}_X, \mathbf{f}_*) = \mathcal{N}(\mathbf{0}, \begin{pmatrix} \mathbf{Q}_{X,X} - \text{diag}(\mathbf{Q}_{X,X} - \mathbf{K}_{X,X}) & \mathbf{Q}_{X,*} \\ \mathbf{Q}_{*,X} & \mathbf{K}_{*,*} \end{pmatrix}) \quad (18.129)$$

The predictive distribution for a single test case is given by

$$q_{\text{fitc}}(f_* | \mathbf{y}) = \mathcal{N}(f_* | \mathbf{k}_{*,Z} \mathbf{\Sigma} \mathbf{K}_{Z,X} \mathbf{\Lambda}^{-1} \mathbf{y}, k_{**} - q_{**} + \mathbf{k}_{*,Z} \mathbf{\Sigma} \mathbf{k}_{Z,*}) \quad (18.130)$$

where $\mathbf{\Lambda} \triangleq \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X} + \sigma^2 \mathbf{I}_N)$, and $\mathbf{\Sigma} \triangleq (\mathbf{K}_{Z,Z} + \mathbf{K}_{Z,X} \mathbf{\Lambda}^{-1} \mathbf{K}_{X,Z})^{-1}$. If we have a batch of test cases, we can assume they are conditionally independent (an approach known as **fully independent conditional** or **FIC**), and multiply the above equation.

The computational cost is the same as for SOR and DTC, but the approach avoids some of the pathologies due to a non-degenerate kernel. In particular, one can show that the FIC method is equivalent to exact GP inference with the following non-degenerate kernel:

$$\mathcal{K}_{\text{fic}}(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i = j \\ \mathcal{K}_{\text{SOR}}(\mathbf{x}_i, \mathbf{x}_j) & \text{if } i \neq j \end{cases} \quad (18.131)$$

18.5.3.4 Learning the inducing points

So far, we have not specified how to choose the inducing points or pseudoinputs \mathbf{Z} . We can treat these like kernel hyperparameters, and choose them so as to maximize the log marginal likelihood, given by

$$\log q(\mathbf{y} | \mathbf{X}, \mathbf{Z}) = \log \int \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{f}_Z) p(\mathbf{f}_Z | \mathbf{Z}) d\mathbf{f}_Z d\mathbf{f}_X \quad (18.132)$$

$$= \log \int p(\mathbf{y} | \mathbf{f}_X) q(\mathbf{f}_X | \mathbf{X}, \mathbf{Z}) d\mathbf{f}_X \quad (18.133)$$

$$= -\frac{1}{2} \log |\mathbf{Q}_{X,X} + \mathbf{\Lambda}| - \frac{1}{2} \mathbf{y}^\top (\mathbf{Q}_{X,X} + \mathbf{\Lambda})^{-1} \mathbf{y} - \frac{n}{2} \log(2\pi) \quad (18.134)$$

where the definition of $\mathbf{\Lambda}$ depends on the method, namely $\mathbf{\Lambda}_{\text{SOR}} = \mathbf{\Lambda}_{\text{dte}} = \sigma^2 \mathbf{I}_N$, and $\mathbf{\Lambda}_{\text{fitc}} = \text{diag}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) + \sigma^2 \mathbf{I}_N$.

If the input domain is \mathbb{R}^d , we can optimize $\mathbf{Z} \in \mathbb{R}^{Md}$ using gradient methods. However, one of the appeals of kernel methods is that they can handle structured inputs, such as strings and graphs (see Section 18.2.3). In this case, we cannot use gradient methods to select the inducing points. A simple approach is to select the inducing points from the training set, as in the subset of data approach in Section 18.5.1, or using the efficient selection mechanism in [Cao+15]. However, we can also use discrete optimization methods, such as simulated annealing (Section 12.9.1), as discussed in [For+18a]. See Figure 18.13 for an illustration.

18.5.4 Sparse variational methods

In this section, we discuss a variational approach to GP inference called the **sparse variational GP** or **SVGP** approximation, also known as the **variational free energy** or **VFE** approach [Tit09; Mat+16]. This is similar to the inducing point methods in Section 18.5.3, except it approximates the posterior, rather than approximating the prior. The variational approach can also easily handle non-conjugate likelihoods, as we will see. For more details, see e.g., [BWR16; Lei+20]. (See also [WKS21] for connections between SVGP and the Nyström method.)

To explain the idea behind SVGP/VFE, let us assume, for simplicity, that the function f is defined over a finite set \mathcal{X} of possible inputs, which we partition into three subsets: the training set \mathbf{X} , a set

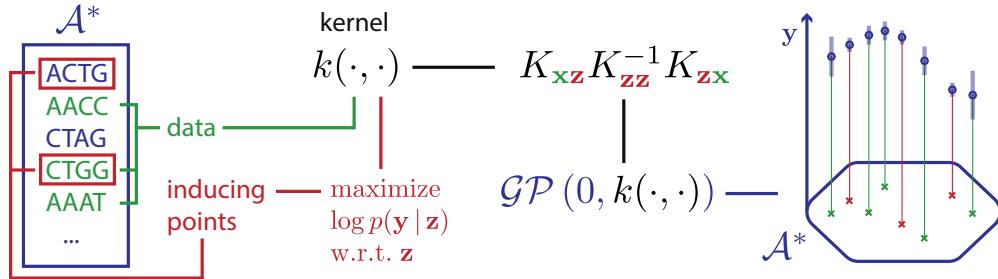


Figure 18.13: Illustration of how to choose inducing points from a discrete input domain (here DNA sequences of length 4) to maximize the log marginal likelihood. From Figure 1 of [For+18a]. Used with kind permission of Vincent Fortuin.

of inducing points \mathbf{Z} , and all other points (which we can think of as the test set), \mathbf{X}_* . (We assume these sets are disjoint.) Let \mathbf{f}_X , \mathbf{f}_Z and \mathbf{f}_* represent the corresponding unknown function values on these points, and let $\mathbf{f} = [\mathbf{f}_X, \mathbf{f}_Z, \mathbf{f}_*]$ be all the unknowns. (Here we work with a fixed-length vector \mathbf{f} , but the result generalizes to Gaussian processes, as explained in [Mat+16].) We assume the function is sampled from a GP, so $p(\mathbf{f}) = \mathcal{N}(\mathbf{m}(\mathcal{X}), \mathcal{K}(\mathcal{X}, \mathcal{X}))$.

The inducing point methods in Section 18.5.3 approximates the GP prior by assuming $p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \approx p(\mathbf{f}_*|\mathbf{f}_Z)p(\mathbf{f}_X|\mathbf{f}_Z)p(\mathbf{f}_Z)$. The inducing points \mathbf{f}_Z are chosen to maximize the likelihood of the observed data. We then perform exact inference in this approximate model. By contrast, in this section, we will keep the model unchanged, but we will instead approximate the posterior $p(\mathbf{f}|\mathbf{y})$ using variational inference.

In the VFE view, the inducing points \mathbf{Z} and inducing variables \mathbf{f}_Z (often denoted by \mathbf{u}) are variational parameters, rather than model parameters, which avoids the risk of overfitting. Furthermore, one can show that as the number of inducing points m increases, the quality of the posterior consistently improves, eventually recovering exact inference. By contrast, in the classical inducing point method, increasing m does not always result in better performance [BWR16].

In more detail, the VFE approach tries to find an approximate posterior $q(\mathbf{f})$ to minimize $D_{\text{KL}}(q(\mathbf{f}) \parallel p(\mathbf{f}|\mathbf{y}))$. The key assumption is that $q(\mathbf{f}) = q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) = p(\mathbf{f}_*, \mathbf{f}_X|\mathbf{f}_Z)q(\mathbf{f}_Z)$, where $p(\mathbf{f}_*, \mathbf{f}_X|\mathbf{f}_Z)$ is computed exactly using the GP prior, and $q(\mathbf{f}_Z)$ is learned, by minimizing $\mathcal{K}(q) = D_{\text{KL}}(q(\mathbf{f}) \parallel p(\mathbf{f}|\mathbf{y}))$.² Intuitively, $q(\mathbf{f}_Z)$ acts as a “bottleneck” which “absorbs” all the observations from \mathbf{y} ; posterior predictions for elements of \mathbf{f}_X or \mathbf{f}_* are then made via their dependence on \mathbf{f}_Z , rather than their dependence on each other.

² One can show that $D_{\text{KL}}(q(\mathbf{f}) \parallel p(\mathbf{f}|\mathbf{y})) = D_{\text{KL}}(q(\mathbf{f}_X, \mathbf{f}_Z) \parallel p(\mathbf{f}_X, \mathbf{f}_Z|\mathbf{y}))$, which is the original objective from [Tit09].

We can derive the form of the loss, which is used to compute the posterior $q(\mathbf{f}_Z)$, as follows:

$$\mathcal{K}(q) = D_{\text{KL}}(q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \| p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z | \mathbf{y})) \quad (18.135)$$

$$= \int q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z) \log \frac{q(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z)}{p(\mathbf{f}_*, \mathbf{f}_X, \mathbf{f}_Z | \mathbf{y})} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.136)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log \frac{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) p(\mathbf{y})}{p(\mathbf{f}_* | \mathbf{f}_X, \mathbf{f}_Z) p(\mathbf{f}_X | \mathbf{f}_Z) p(\mathbf{f}_Z) p(\mathbf{y} | \mathbf{f}_X)} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.137)$$

$$= \int p(\mathbf{f}_*, \mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z) p(\mathbf{y})}{p(\mathbf{f}_Z) p(\mathbf{y} | \mathbf{f}_X)} d\mathbf{f}_* d\mathbf{f}_X d\mathbf{f}_Z \quad (18.138)$$

$$= \int q(\mathbf{f}_Z) \log \frac{q(\mathbf{f}_Z)}{p(\mathbf{f}_Z)} d\mathbf{f}_Z - \int p(\mathbf{f}_X | \mathbf{f}_Z) q(\mathbf{f}_Z) \log p(\mathbf{y} | \mathbf{f}_X) d\mathbf{f}_X d\mathbf{f}_Z + C \quad (18.139)$$

$$= D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) - \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] + C \quad (18.140)$$

where $C = \log p(\mathbf{y})$ is an irrelevant constant.

We can alternatively write the objective as an evidence lower bound that we want to maximize:

$$\log p(\mathbf{y}) = \mathcal{K}(q) + \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.141)$$

$$\geq \mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \triangleq \mathcal{L}(q) \quad (18.142)$$

Now suppose we choose a Gaussian posterior approximation, $q(\mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_Z | \mathbf{m}, \mathbf{S})$. Since $p(\mathbf{f}_Z) = \mathcal{N}(\mathbf{f}_Z | \mathbf{0}, \mathcal{K}(\mathbf{Z}, \mathbf{Z}))$, we can compute the KL term in closed form using the formula for KL divergence between Gaussians (Equation (5.77)). To compute the expected log-likelihood term, we first need to compute the induced posterior over the latent function values at the training points:

$$q(\mathbf{f}_X | \mathbf{m}, \mathbf{S}) = \int p(\mathbf{f}_X | \mathbf{f}_Z, \mathbf{X}, \mathbf{Z}) q(\mathbf{f}_Z | \mathbf{m}, \mathbf{S}) d\mathbf{f}_Z = \mathcal{N}(\mathbf{f}_X | \tilde{\mathbf{\mu}}, \tilde{\mathbf{\Sigma}}) \quad (18.143)$$

$$\tilde{\mathbf{\mu}}_i = m(\mathbf{x}_i) + \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathbf{m} - m(\mathbf{Z})) \quad (18.144)$$

$$\tilde{\Sigma}_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) - \boldsymbol{\alpha}(\mathbf{x}_i)^T (\mathcal{K}(\mathbf{Z}, \mathbf{Z}) - \mathbf{S}) \boldsymbol{\alpha}(\mathbf{x}_j) \quad (18.145)$$

$$\boldsymbol{\alpha}(\mathbf{x}_i) = \mathcal{K}(\mathbf{Z}, \mathbf{Z})^{-1} \mathcal{K}(\mathbf{Z}, \mathbf{x}_i) \quad (18.146)$$

Hence the marginal at a single point is $q(f_n) = \mathcal{N}(f_n | \tilde{\mu}_n, \tilde{\Sigma}_{nn})$, which we can use to compute the expected log likelihood:

$$\mathbb{E}_{q(\mathbf{f}_X)} [\log p(\mathbf{y} | \mathbf{f}_X)] = \sum_{n=1}^N \mathbb{E}_{q(f_n)} [\log p(y_n | f_n)] \quad (18.147)$$

We discuss how to compute these expectations below.

18.5.4.1 Gaussian likelihood

If we have a Gaussian observation model, we can compute the expected log likelihood in closed form. In particular, if we assume $m(\mathbf{x}) = \mathbf{0}$, we have

$$\mathbb{E}_{q(f_n)} [\log \mathcal{N}(y_n | f_n, \beta^{-1})] = \log \mathcal{N}(y_n | \mathbf{k}_n^T \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1}) - \frac{1}{2} \beta \tilde{k}_{nn} - \frac{1}{2} \text{tr}(\mathbf{S} \mathbf{\Lambda}_n) \quad (18.148)$$

where $\tilde{k}_{nn} = k_{nn} - \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1} \mathbf{k}_n$, \mathbf{k}_n is the n 'th column of $\mathbf{K}_{Z,X}$ and $\boldsymbol{\Lambda}_n = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{k}_n \mathbf{k}_n^\top \mathbf{K}_{Z,Z}^{-1}$.

Hence the overall ELBO has the form

$$\mathcal{L}(q) = \log \mathcal{N}(\mathbf{y} | \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{m}, \beta^{-1} \mathbf{I}_N) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{S} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}) \quad (18.149)$$

$$- \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.150)$$

where $\mathbf{Q}_{X,X} = \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X}$.

To compute the gradients of this, we leverage the following result [OA09]:

$$\frac{\partial}{\partial \mu} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)}[h(x)] = \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)} \left[\frac{\partial}{\partial x} h(x) \right] \quad (18.151)$$

$$\frac{\partial}{\partial \sigma^2} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)}[h(x)] = \frac{1}{2} \mathbb{E}_{\mathcal{N}(x|\mu, \sigma^2)} \left[\frac{\partial^2}{\partial x^2} h(x) \right] \quad (18.152)$$

We then substitute $h(x)$ with $\log p(y_n|f_n)$. Using this, one can show

$$\nabla_{\mathbf{m}} \mathcal{L}(q) = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} - \boldsymbol{\Lambda} \mathbf{m} \quad (18.153)$$

$$\nabla_{\mathbf{S}} \mathcal{L}(q) = \frac{1}{2} \mathbf{S}^{-1} - \frac{1}{2} \boldsymbol{\Lambda} \quad (18.154)$$

Setting the derivatives to zero gives the optimal solution:

$$\mathbf{S} = \boldsymbol{\Lambda}^{-1} \quad (18.155)$$

$$\boldsymbol{\Lambda} = \beta \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} + \mathbf{K}_{Z,Z}^{-1} \quad (18.156)$$

$$\mathbf{m} = \beta \boldsymbol{\Lambda}^{-1} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} \mathbf{y} \quad (18.157)$$

This is called **sparse GP regression** or **SGPR** [Tit09].

With these parameters, the lower bound on the log marginal likelihood is given by

$$\log p(\mathbf{y}) \geq \log \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1} \mathbf{K}_{Z,X} + \beta^{-1} \mathbf{I}) - \frac{1}{2} \beta \text{tr}(\mathbf{K}_{X,X} - \mathbf{Q}_{X,X}) \quad (18.158)$$

(This is called the ‘‘collapsed’’ lower bound, since we have marginalized out \mathbf{f}_Z .) If $Z = X$, then $\mathbf{K}_{Z,Z} = \mathbf{K}_{Z,X} = \mathbf{K}_{X,X}$, so the bound becomes tight, and we have $\log p(\mathbf{y}) = \log \mathcal{N}(\mathbf{y} | \mathbf{0}, \mathbf{K}_{X,X} + \beta^{-1} \mathbf{I})$.

Equation (18.158) is almost the same as the log marginal likelihood for the DTC model in Equation (18.134), except for the trace term; it is this latter term that prevents overfitting, due to the fact that we treat \mathbf{f}_Z as variational parameters of the posterior rather than model parameters of the prior.

18.5.4.2 Non-Gaussian likelihood

In this section, we briefly consider the case of non-Gaussian likelihoods, which arise when using GPs for classification or for count data (see Section 18.4). We can compute the gradients of the expected log likelihood by defining $h(f_n) = \log p(y_n|f_n)$ and then using a Monte Carlo approximation to Equation (18.151) and Equation (18.152). In the case of a binary classifier, we can use the results in Table 18.2 to compute the inner $\frac{\partial}{\partial f_n} h(f_n)$ and $\frac{\partial^2}{\partial f_n^2} h(f_n)$ terms. Alternatively, we can use numerical integration techniques, such as those discussed in Section 8.5.1.4. (See also [WSS21].)

18.5.4.3 Minibatch SVI

Computing the optimal variational solution in Section 18.5.4.1 requires solving a batch optimization problem, which takes $O(M^3 + NM^2)$ time. This may still be too slow if N is large, unless M is small, which compromises accuracy.

An alternative approach is to perform stochastic optimization of the VFE objective, instead of batch optimization. This is known as stochastic variational inference (see Section 10.1.4). The key observation is that the log likelihood in Equation (18.147) is a sum of N terms, which we can approximate with minibatch sampling to compute noisy estimates of the gradient, as proposed in [HFL13].

In more detail, the objective becomes

$$\mathcal{L}(q) = \left[\frac{N}{B} \sum_{b=1}^B \frac{1}{|\mathcal{B}_b|} \sum_{n \in \mathcal{B}_b} \mathbb{E}_{q(f_n)} [\log p(y_n | f_n)] \right] - D_{\text{KL}}(q(\mathbf{f}_Z) \| p(\mathbf{f}_Z)) \quad (18.159)$$

where \mathcal{B}_b is the b 'th batch, and B is the number of batches. Since the GP model (with Gaussian likelihoods) is in the exponential family, we can efficiently compute the natural gradient (Section 6.4) of Equation (18.159) wrt the canonical parameters of $q(\mathbf{f}_Z)$; this converges much faster than following the standard gradient. See [HFL13] for details.

18.5.5 Exploiting parallelization and structure via kernel matrix multiplies

It takes $O(N^3)$ time to compute the Cholesky decomposition of $\mathbf{K}_{X,X}$, which is needed to solve the linear system $\mathbf{K}_\sigma \boldsymbol{\alpha} = \mathbf{y}$ and to compute $|\mathbf{K}_{X,X}|$. An alternative to Cholesky decomposition is to use linear algebra methods, often called **Krylov subspace methods** based just on **matrix vector multiplication** or **MVM**. These approaches are often much faster.

In short, if the kernel matrix $\mathbf{K}_{X,X}$ has special algebraic structure, which is often the case through either the choice of kernel or the structure of the inputs, then it is typically easier to exploit this structure in performing fast matrix multiplies. Moreover, even if the kernel matrix **does not** have special structure, matrix multiplies are trivial to parallelize, and can thus be greatly accelerated by GPUs, unlike Cholesky based methods which are largely sequential. Algorithms based on matrix multiplies are in harmony with modern hardware advances, which enable significant parallelization.

18.5.5.1 Using conjugate gradient and Lanczos methods

We can solve the linear system $\mathbf{K}_\sigma \boldsymbol{\alpha} = \mathbf{y}$ using conjugate gradients (CG). The key computational step in CG is the ability to perform MVMs. Let $\tau(\mathbf{K}_\sigma)$ be the time complexity of a single MVM with \mathbf{K}_σ . For a dense $n \times n$ matrix, we have $\tau(\mathbf{K}_\sigma) = n^2$; however, we can speed this up if \mathbf{K}_σ is sparse or structured, as we discuss below.

Even if \mathbf{K}_σ is dense, we may still be able to save time by solving the linear system approximately. In particular, if we perform C iterations, CG will take $O(C\tau(\mathbf{K}_\sigma))$ time. If we run for $C = n$, and $\tau(\mathbf{K}_\sigma) = n^2$, it gives the exact solution in $O(n^3)$ time. However, often we can use fewer iterations and still get good accuracy, depending on the condition number of \mathbf{K}_σ .

We can compute the log determinant of a matrix using the MVM primitive with a similar iterative method known as **stochastic Lanczos quadrature** [UCS17; Don+17a]. This takes $O(L\tau(\mathbf{K}_\sigma))$ time for L iterations.

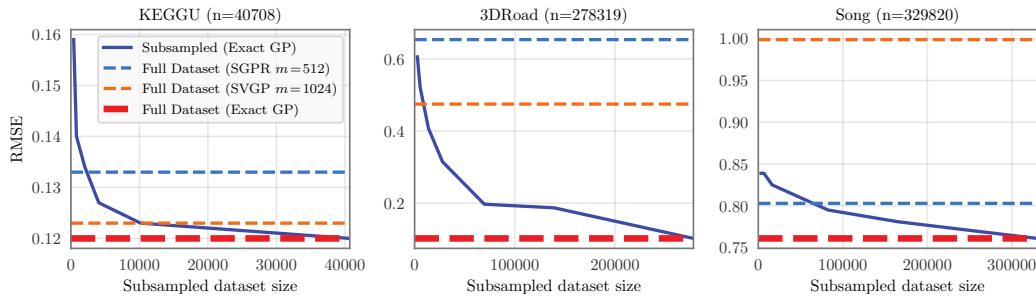


Figure 18.14: RMSE on test set as a function of training set size using a GP with Matern 3/2 kernel with shared lengthscale across all dimensions. Solid lines: exact inference. Dashed blue: SGPR method (closed-form batch solution to the Gaussian variational approximation) of Section 18.5.4.1 with $M = 512$ inducing points. Dashed orange: SVGP method (SGD on Gaussian variational approximation) of Section 18.5.4.3 with $M = 1024$ inducing points. Number of input dimensions: KEGGU $D = 27$, 3DRoad $D = 3$, Song $D = 90$. From Figure 4 of [Wan+19a]. Used with kind permission of Andrew Wilson.

These methods have been used in the **blackbox matrix-matrix multiplication** (BBMM) inference procedure of [Gar+18a], which formulates a batch approach to CG that can be effectively parallelized on GPUs. Using 8 GPUs, this enabled the authors of [Wan+19a] to perform exact inference for a GP regression model on $N \sim 10^4$ datapoints in seconds, $N \sim 10^5$ datapoints in minutes, and $N \sim 10^6$ datapoints in hours.

Interestingly, Figure 18.14 shows that exact GP inference on a subset of the data can often outperform approximate inference on the full data. We also see that performance of exact GPs continues to significantly improve as we increase the size of the data, suggesting that GPs are not only useful in the small-sample setting. In particular, the BBMM is an exact method, and so will preserve the non-parametric representation of a GP with a non-degenerate kernel. By contrast, standard scalable approximations typically operate by replacing the exact kernel with an approximation that corresponds to a parametric model. The non-parametric GPs are able to grow their capacity with more data, benefiting more significantly from the structure present in large datasets.

18.5.5.2 Kernels with compact support

Suppose we use a kernel with **compact support**, where $\mathcal{K}(\mathbf{x}, \mathbf{x}') = 0$ if $\|\mathbf{x} - \mathbf{x}'\| > \epsilon$ for some threshold ϵ (see e.g., [MR09]), then \mathbf{K}_σ will be sparse, so $\tau(\mathbf{K}_\sigma)$ will be $O(N)$. We can also induce sparsity and structure in other ways, as we discuss in Section 18.5.5.3.

18.5.5.3 KISS

One way to ensure that MVMs are fast is to force the kernel matrix to have structure. The **structured kernel interpolation** (SKI) method of [WN15] does this as follows. First it assumes we have a set of inducing points, with Gram matrix $\mathbf{K}_{Z,Z}$. It then interpolates these values to predict the entries of the full kernel matrix using

$$\mathbf{K}_{X,X} \approx \mathbf{W}_X \mathbf{K}_{Z,Z} \mathbf{W}_X^\top \quad (18.160)$$

where \mathbf{W}_X is a sparse matrix containing interpolation weights. If we use cubic interpolation, each row only has 4 nonzeros. Thus we can compute $(\mathbf{W}_X \mathbf{K}_{Z,Z} \mathbf{W}_X^\top) \mathbf{v}$ for any vector \mathbf{v} in $O(N + M^2)$ time.

Note that the **SKI** approach generalizes all inducing point methods. For example, we can recover the subset of regressors method (SOR) method by setting the interpolation weights to $\mathbf{W} = \mathbf{K}_{X,Z} \mathbf{K}_{Z,Z}^{-1}$. We can identify this procedure as performing a global Gaussian process interpolation strategy on the user specified kernel. See [WN15] and [WDN15] for more details.

In 1d, we can further reduce the running time by choosing the inducing points to be on a regular grid, so that $\mathbf{K}_{Z,Z}$ is a Toeplitz matrix. In higher dimensions, we need to use a multidimensional grid of points, resulting in $\mathbf{K}_{Z,Z}$ being a Kronecker product of Toeplitz matrices. This enables matrix vector multiplication in $O(N + M \log M)$ time and $O(N + M)$ space. The resulting method is called **KISS-GP** [WN15], which stands for “kernel interpolation for scalable, structured GPs”.

Unfortunately, the KISS method can take exponential time in the input dimensions D when exploiting Kronecker structure in $\mathbf{K}_{Z,Z}$, due to the need to create a fully connected multidimensional lattice. In [Gar+18b], they propose a method called **SKIP**, which stands for “SKI for products”. The idea is to leverage the fact that many kernels (including ARD) can be written as a product of 1d kernels: $\mathcal{K}(\mathbf{x}, \mathbf{x}') = \prod_{d=1}^D \mathcal{K}^d(\mathbf{x}, \mathbf{x}')$. This can be combined with the 1d SKI method to enable fast MVMs. The overall running time to compute the log marginal likelihood (which is the bottleneck for kernel learning) using C iterations of CG and a Lanczos decomposition of rank L , becomes $O(DL(N + M \log M) + L^3 N \log D + CL^2 N)$. Typical values are $L \sim 10^1$ and $C \sim 10^2$.

18.5.5.4 Tensor train methods

Consider the Gaussian VFE approach in Section 18.5.4. We have to estimate the covariance \mathbf{S} and the mean \mathbf{m} . We can represent \mathbf{S} efficiently using Kronecker structure, as used by KISS. Additionally, we can represent \mathbf{m} efficiently using the **tensor train decomposition** [Ose11] in combination with **SKI** [WN15]. The resulting **TT-GP** method can scale efficiently to billions of inducing points, as explained in [INK18].

18.5.6 Converting a GP to an SSM

Consider a function defined on a 1d scalar input, such as a time index. For many stationary 1d kernels, the corresponding GP can be modeled using a linear time invariant (LTI) stochastic differential equation (SDE)³; this SDE can then be converted to a linear-Gaussian state space model (Section 29.1) as first proposed in [HS10]. For example, consider the exponential kernel in Equation (18.14), $\mathcal{K}(t, t') = \frac{q}{2\lambda} \exp(-\lambda|t - t'|)$, which corresponds to a Matérn kernel with $\nu = 1/2$. The corresponding SDE is the Ornstein-Uhlenbeck process which has the form $\frac{dx(t)}{dt} = -\lambda x(t) + w(t)$, where $w(t)$ is a white noise process with spectral density q [SS19, p258].⁴ For other kernels (such as Matérn with $\nu = 3/2$), we need to use multiple latent states in order to capture higher order

3. The condition is that the spectral density of the covariance function has to be a rational function. This includes many kernels, such as the Matérn kernel, but excludes the squared exponential (RBF) kernel. However the latter can be approximated by an SDE, as explained in [SS19, p261].

4. This is sometimes written as $dx = -\lambda x dt + d\beta$, where $\beta(t)$ is a Brownian noise process, and $w(t) = \frac{d\beta(t)}{dt}$, as explained in [SS19, p45].

derivative terms (see [Supplementary](#) Section 18.2 for details). Furthermore, for higher dimensional inputs, we need to use even more latent states, to enforce the Markov property [[DSP21](#)].

Once we have converted the GP to LG-SSM form, we can perform exact inference in $O(N)$ time using Kalman smoothing, as explained in [Section 8.2.3](#). Furthermore, if we have access to a highly parallel processor, such as a GPU, we can reduce the time to $\log(N)$ [[CZS22](#)], as explained in [Section 8.2.3.4](#).

18.6 Learning the kernel

In [[Mac98](#)], David MacKay asked: “How can Gaussian processes replace neural networks? Have we thrown the baby out with the bathwater?” This remark was made in the late 1990s, at the end of the second wave of neural networks. Researchers and practitioners had grown weary of the design decisions associated with neural networks — such as activation functions, optimization procedures, architecture design — and the lack of a principled framework to make these decisions. Gaussian processes, by contrast, were perceived as flexible and principled probabilistic models, which naturally followed from Radford Neal’s results on infinite neural networks [[Nea96](#)], which we discuss in more depth in [Section 18.7](#).

However, MacKay [[Mac98](#)] noted that neural networks could discover rich representations of data through adaptive hidden basis functions, while Gaussian processes with standard kernel functions, such as the RBF kernel, are essentially just smoothing devices. Indeed, the generalization properties of Gaussian processes hinge on the suitability of the kernel function. *Learning* the kernel is how we do representation learning with Gaussian processes, and in many cases will be crucial for good performance — especially when we wish to perform extrapolation, making predictions far away from the data [[WA13](#); [Wil+14](#)].

As we will see, learning a kernel is in many ways analogous to training a neural network. Moreover, neural networks and Gaussian processes can be synergistically combined through approaches such as deep kernel learning (see [Section 18.6.6](#)) and NN-GPs ([Section 18.7.2](#)).

18.6.1 Empirical Bayes for the kernel parameters

Suppose, as in [Section 18.3.2](#), we are performing 1d regression using a GP with an RBF kernel. Since the data has observation noise, the kernel has the following form:

$$\mathcal{K}_y(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x_p - x_q)^2\right) + \sigma_y^2 \delta_{pq} \quad (18.161)$$

Here ℓ is the horizontal scale over which the function changes, σ_f^2 controls the vertical scale of the function, and σ_y^2 is the noise variance. Figure 18.15 illustrates the effects of changing these parameters. We sampled 20 noisy datapoints from the SE kernel using $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and then made predictions various parameters, conditional on the data. In Figure 18.15(a), we use $(\ell, \sigma_f, \sigma_y) = (1, 1, 0.1)$, and the result is a good fit. In Figure 18.15(b), we increase the length scale to $\ell = 3$; now the function looks smoother, but we are arguably underfitting.

To estimate the kernel parameters θ (sometimes called hyperparameters), we could use exhaustive search over a discrete grid of values, with validation loss as an objective, but this can be quite slow. (This is the approach used by nonprobabilistic methods, such as SVMs, to tune kernels.) Here we

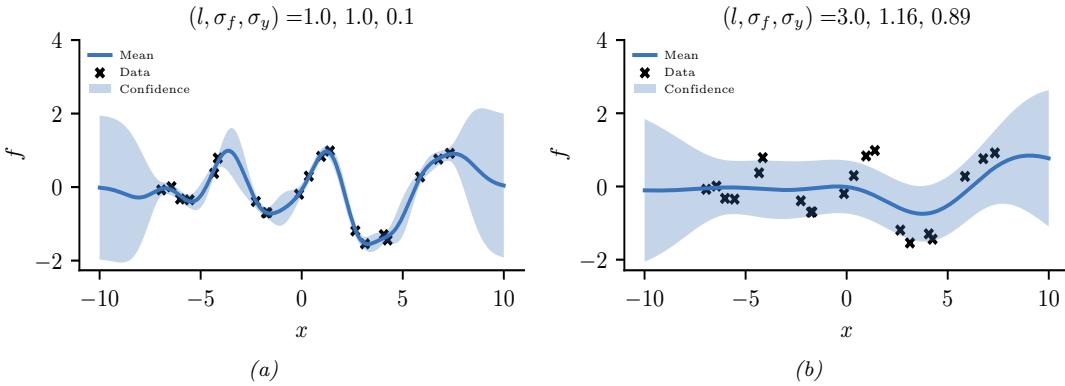


Figure 18.15: Some 1d GPs with RBF kernels but different hyper-parameters fit to 20 noisy observations. The hyper-parameters $(\ell, \sigma_f, \sigma_y)$ are as follows: (a) $(1, 1, 0.1)$ (b) $(3.0, 1.16, 0.89)$. Adapted from Figure 2.5 of [RW06]. Generated by `gpr_demo_change_hparams.ipynb`.

consider an empirical Bayes approach, which will allow us to use continuous optimization methods, which are much faster. In particular, we will maximize the marginal likelihood

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X})p(\mathbf{f}|\mathbf{X}, \boldsymbol{\theta})d\mathbf{f} \quad (18.162)$$

(The reason it is called the marginal likelihood, rather than just likelihood, is because we have marginalized out the latent Gaussian vector \mathbf{f} .) Since $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K})$, and $p(\mathbf{y}|\mathbf{f}) = \prod_{n=1}^N \mathcal{N}(y_n|f_n, \sigma_y^2)$, the marginal likelihood is given by

$$\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_\sigma) = -\frac{1}{2}\mathbf{y}\mathbf{K}_\sigma^{-1}\mathbf{y} - \frac{1}{2}\log |\mathbf{K}_\sigma| - \frac{N}{2}\log(2\pi) \quad (18.163)$$

where the dependence of \mathbf{K}_σ on $\boldsymbol{\theta}$ is implicit. The first term is a data fit term, the second term is a model complexity term, and the third term is just a constant. To understand the tradeoff between the first two terms, consider a SE kernel in 1d, as we vary the length scale ℓ and hold σ_y^2 fixed. Let $J(\ell) = -\log p(\mathbf{y}|\mathbf{X}, \ell)$. For short length scales, the fit will be good, so $\mathbf{y}^\top \mathbf{K}_\sigma^{-1} \mathbf{y}$ will be small. However, the model complexity will be high: \mathbf{K} will be almost diagonal, since most points will not be considered “near” any others, so the $\log |\mathbf{K}_\sigma|$ will be large. For long length scales, the fit will be poor but the model complexity will be low: \mathbf{K} will be almost all 1’s, so $\log |\mathbf{K}_\sigma|$ will be small.

We now discuss how to maximize the marginal likelihood. One can show that

$$\frac{\partial}{\partial \theta_j} \log p(\mathbf{y} | \mathbf{X}, \boldsymbol{\theta}) = \frac{1}{2} \mathbf{y}^\top \mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j} \mathbf{K}_\sigma^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_\sigma^{-1} \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j}) \quad (18.164)$$

$$= \frac{1}{2} \text{tr} \left((\boldsymbol{\alpha}\boldsymbol{\alpha}^\top - \mathbf{K}_\sigma^{-1}) \frac{\partial \mathbf{K}_\sigma}{\partial \theta_j} \right) \quad (18.165)$$

where $\alpha = \mathbf{K}_\sigma^{-1} \mathbf{y}$. It takes $O(N^3)$ time to compute \mathbf{K}_σ^{-1} , and then $O(N^2)$ time per hyper-parameter to compute the gradient.

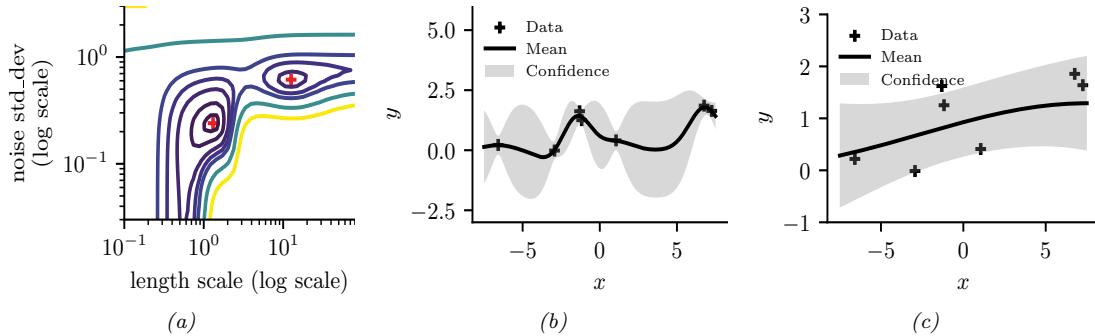


Figure 18.16: Illustration of local minima in the marginal likelihood surface. (a) We plot the log marginal likelihood vs σ_y^2 and ℓ , for fixed $\sigma_f^2 = 1$, using the 7 datapoints shown in panels b and c. (b) The function corresponding to the lower left local minimum, $(\ell, \sigma_n^2) \approx (1, 0.2)$. This is quite “wiggly” and has low noise. (c) The function corresponding to the top right local minimum, $(\ell, \sigma_n^2) \approx (10, 0.8)$. This is quite smooth and has high noise. The data was generated using $(\ell, \sigma_n^2) = (1, 0.1)$. Adapted from Figure 5.5 of [RW06]. Generated by [gpr_demo_marglik.ipynb](#).

The form of $\frac{\partial \mathbf{K}_x}{\partial \theta_j}$ depends on the form of the kernel, and which parameter we are taking derivatives with respect to. Often we have constraints on the hyper-parameters, such as $\sigma_y^2 \geq 0$. In this case, we can define $\theta = \log(\sigma_y^2)$, and then use the chain rule.

Given an expression for the log marginal likelihood and its derivative, we can estimate the kernel parameters using any standard gradient-based optimizer. However, since the objective is not convex, local minima can be a problem, as we illustrate below, so we may need to use multiple restarts.

18.6.1.1 Example

Consider Figure 18.16. We use the SE kernel in Equation (18.161) with $\sigma_f^2 = 1$, and plot $\log p(\mathbf{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where \mathbf{X} and \mathbf{y} are the 7 datapoints shown in panels b and c as we vary ℓ and σ_y^2). The two local optima are indicated by + in panel a. The bottom left optimum corresponds to a low-noise, short-length scale solution (shown in panel b). The top right optimum corresponds to a high-noise, long-length scale solution (shown in panel c). With only 7 datapoints, there is not enough evidence to confidently decide which is more reasonable, although the more complex model (panel b) has a marginal likelihood that is about 60% higher than the simpler model (panel c). With more data, the more complex model would become even more preferred.

Figure 18.16 illustrates some other interesting (and typical) features. The region where $\sigma_y^2 \approx 1$ (top of panel a) corresponds to the case where the noise is very high; in this regime, the marginal likelihood is insensitive to the length scale (indicated by the horizontal contours), since all the data is explained as noise. The region where $\ell \approx 0.5$ (left hand side of panel a) corresponds to the case where the length scale is very short; in this regime, the marginal likelihood is insensitive to the noise level (indicated by the vertical contours), since the data is perfectly interpolated. Neither of these regions would be chosen by a good optimizer.

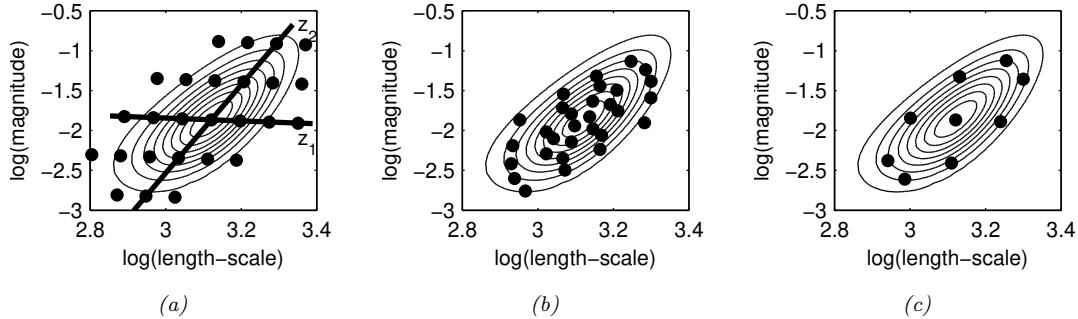


Figure 18.17: Three different approximations to the posterior over hyper-parameters: grid-based, Monte Carlo, and central composite design. From Figure 3.2 of [Van10]. Used with kind permission of Jarno Vanhatalo.

18.6.2 Bayesian inference for the kernel parameters

When we have a small number of datapoints (e.g., when using GPs for blackbox optimization, as we discuss in Section 6.6), using a point estimate of the kernel parameters can give poor results [Bul11; WF14]. As a simple example, if the function values that have been observed so far are all very similar, then we may estimate $\hat{\sigma} \approx 0$, which will result in overly confident predictions.⁵

To overcome such overconfidence, we can compute a posterior over the kernel parameters. If the dimensionality of θ is small, we can compute a discrete grid of possible values, centered on the MAP estimate $\hat{\theta}$ (computed as above). We can then approximate the posterior using

$$p(\mathbf{f}|\mathcal{D}) = \sum_{s=1}^S p(\mathbf{f}|\mathcal{D}, \theta_s) p(\theta_s|\mathcal{D}) w_s \quad (18.166)$$

where w_s denotes the weight for grid point s .

In higher dimensions, a regular grid suffers from the curse of dimensionality. One alternative is place grid points at the mode, and at a distance $\pm 1\text{sd}$ from the mode along each dimension, for a total of $2|\theta| + 1$ points. This is called a **central composite design** [RMC09]. See Figure 18.17 for an illustration.

In higher dimensions, we can use Monte Carlo inference for the kernel parameters when computing Equation (18.166). For example, [MA10] shows how to use slice sampling (Section 12.4.1) for this task, [Hen+15] shows how to use HMC (Section 12.5), and [BBV11a] shows how to use SMC (Chapter 13).

In Figure 18.18, we illustrate the difference between kernel optimization vs kernel inference. We fit a 1d dataset using a kernel of the form

$$\mathcal{K}(r) = \sigma_1^2 \mathcal{K}_{\text{SE}}(r; \tau) \mathcal{K}_{\cos}(r; \rho_1) + \sigma_2^2 \mathcal{K}_{32}(r; \rho_2) \quad (18.167)$$

where $\mathcal{K}_{\text{SE}}(r; \ell)$ is the squared exponential kernel (Equation (18.12)), $\mathcal{K}_{\cos}(r; \rho_1)$ is the cosine kernel (Equation (18.19)), and $\mathcal{K}_{32}(r; \rho_2)$ is the Matérn $\frac{3}{2}$ kernel (Equation (18.15)). We then compute a

⁵. In [WSN00; BBV11b], they show how we can put a conjugate prior on σ^2 and integrate it out, to generate a Student version of the GP, which is more robust.

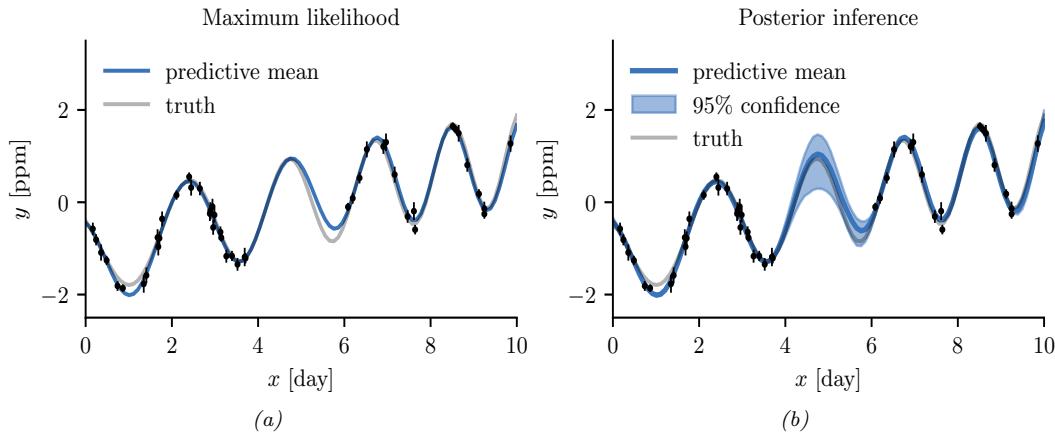


Figure 18.18: Difference between estimation and inference for kernel hyper-parameters. (a) Empirical Bayes approach based on optimization. We plot the posterior predicted mean given a plug-in estimate, $\mathbb{E}[f(x)|\mathcal{D}, \hat{\theta}]$. (b) Bayesian approach based on HMC. We plot the posterior predicted mean, marginalizing over hyper-parameters, $\mathbb{E}[f(x)|\mathcal{D}]$. Generated by [gp_kernel_opt.ipynb](#).

point-estimate of the kernel parameters using empirical Bayes, and posterior samples using HMC. We then predicting the posterior mean of f on a 1d test set by plugging in the MLE or averaging over samples. We see that the latter captures more uncertainty (beyond the uncertainty captured by the Gaussian itself).

18.6.3 Multiple kernel learning for additive kernels

A special case of kernel learning arises when the kernel is a sum of B base kernels

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{x}, \mathbf{x}') \quad (18.168)$$

Optimizing the weights $w_b > 0$ using structural risk minimization is known as **multiple kernel learning**; see e.g., [Rak+08] for details.

Now suppose we constrain the base kernels to depend on a subset of the variables. Furthermore, suppose we enforce a hierarchical inclusion property (e.g., including the kernel k_{123} means we must also include k_{12} , k_{13} and k_{23}), as illustrated in Figure 18.19(left). This is called **hierarchical kernel learning**. We can find a good subset from this model class using convex optimization [Bac09]; however, this requires the use of cross validation to estimate the weights. A more efficient approach is to use the empirical Bayes approach described in [DNR11].

In many cases, it is common to restrict attention to first order additive kernels:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^D \mathcal{K}_d(x_d, x'_d) \quad (18.169)$$

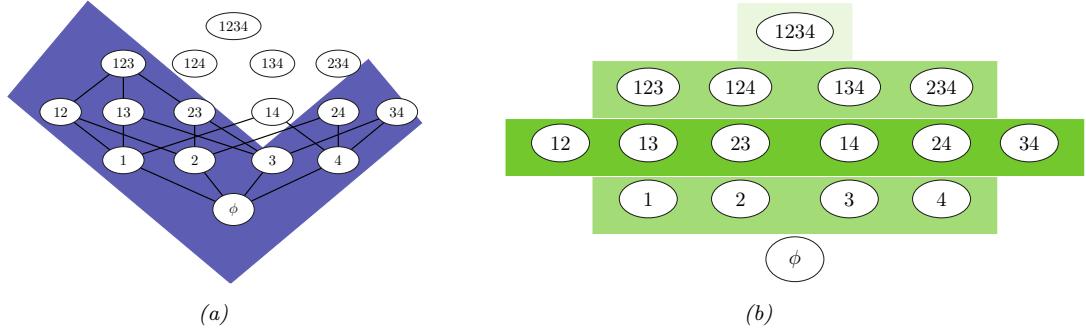


Figure 18.19: Comparison of different additive model classes for a 4d function. Circles represent different interaction terms, ranging from first-order to fourth-order. Left: hierarchical kernel learning uses a nested hierarchy of terms. Right: additive GPs use a weighted sum of additive kernels of different orders. Color shades represent different weighting terms. Adapted from Figure 6.2 of [Duv14].

The resulting function then has the form

$$f(\mathbf{x}) = f_1(x_1) + \dots + f_D(x_D) \quad (18.170)$$

This is called a **generalized additive model** or **GAM**.

Figure 18.20 shows an example of this, where each base kernel has the form $\mathcal{K}_d(x_d, x'_d) = \sigma_d^2 \text{SE}(x_d, x'_d | \ell_d)$. In Figure 18.20, we see that the σ_d^2 terms for the coarse and fine features are set to zero, indicating that these inputs have no impact on the response variable.

[DBW20] considers additive kernels operating on different linear projections of the inputs:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \sum_{b=1}^B w_b \mathcal{K}_b(\mathbf{P}_b \mathbf{x}, \mathbf{P}_b \mathbf{x}') \quad (18.171)$$

Surprisingly, they show that these models can match or exceed the performance of kernels operating on the original space, even when the projections are into a **single** dimension, and not learned. In other words, it is possible to reduce many regression problems to a single dimension without loss in performance. This finding is particularly promising for scalable inference, such as KISS (see Section 18.5.5.3), and active learning, which are greatly simplified in a low dimensional setting.

More recently, [LBH22] has proposed the **orthogonal additive kernel** (OAK), which imposes an orthogonality constraint on the additive functions. This ensures an identifiable, low-dimensional representation of the functional relationship, and results in improved performance.

18.6.4 Automatic search for compositional kernels

Although the above methods can estimate the hyperparameters of a specified set of kernels, they do not choose the kernels themselves (other than the special case of selecting a subset of kernels from a set). In this section, we describe a method, based on [Duv+13], for sequentially searching through the space of increasingly complex GP models so as to find a parsimonious description of the data. (See also [BHB22] for a review.)

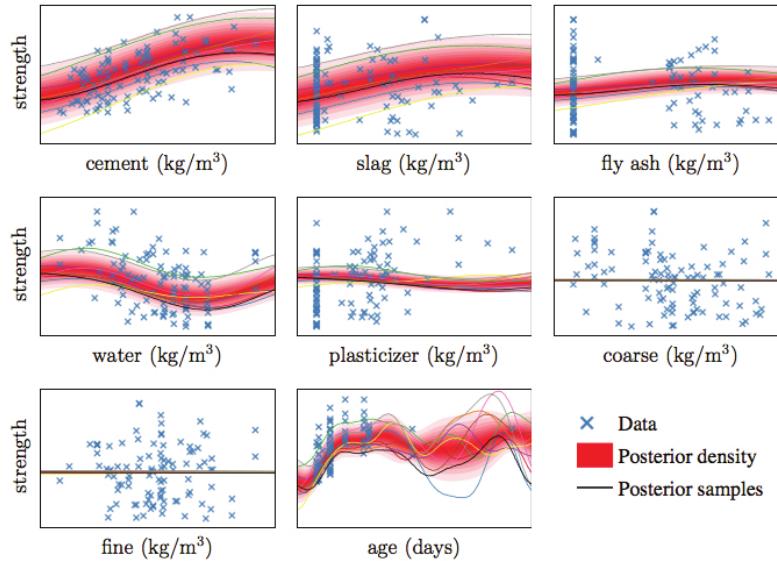


Figure 18.20: Predictive distribution of each term in a GP-GAM model applied to a dataset with 8 continuous inputs and 1 continuous output, representing the strength of some concrete. From Figure 2.7 of [Duv14]. Used with kind permission of David Duvenaud.

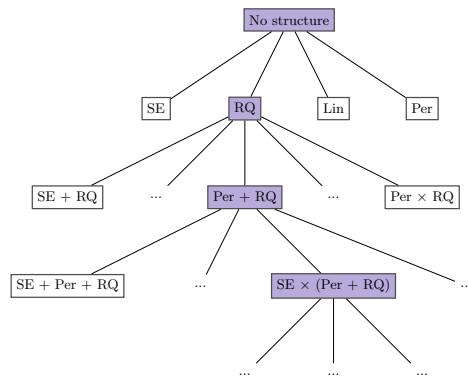


Figure 18.21: Example of a search tree over kernel expressions. Adapted from Figure 3.2 of [Duv14].

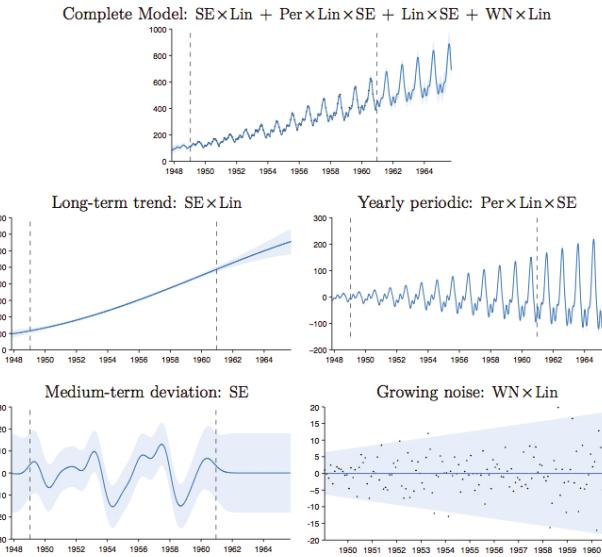


Figure 18.22: Top row: airline dataset and posterior distribution of the model discovered after a search of depth 10. Subsequent rows: predictions of the individual components. From Figure 3.5 of [Duv14], based on [Llo+14]. Used with kind permission of David Duvenaud.

We start with a simple kernel, such as the white noise kernel, and then consider replacing it with a set of possible alternative kernels, such as an SE kernel, RQ kernel, etc. We use the BIC score (Section 3.8.7.2) to evaluate each candidate model (choice of kernel) m . This has the form $BIC(m) = \log p(\mathcal{D}|m) - \frac{1}{2}|m|\log N$, where $p(\mathcal{D}|m)$ is the marginal likelihood, and $|m|$ is the number of parameters. The first term measures fit to the data, and the second term is a complexity penalty. We can also consider replacing a kernel by the addition of two kernels, $k \rightarrow (k + k')$, or the multiplication of two kernels, $k \rightarrow (k \times k')$. See Figure 18.21 for an illustration of the search space.

Searching through this space is similar to what a human expert would do. In particular, if we find structure in the residuals, such as periodicity, we can propose a certain “move” through the space. We can also start with some structure that is assumed to hold globally, such as linearity, but if we find this only holds locally, we can multiply the kernel by an SE kernel. We can also add input dimensions incrementally, to capture higher order interactions.

Figure 18.22 shows the output of this process applied to a dataset of monthly totals of international airline passengers. The input to the GP is the set of time stamps, $\mathbf{x} = 1 : t$; there are no other features.

The observed data lies in between the dotted vertical lines; curves outside of this region are extrapolations. We see that the system has discovered a fairly interpretable set of patterns in the data. Indeed, it is possible to devise an algorithm to automatically convert the output of this search process to a natural language summary, as shown in [Llo+14]. In this example, it summarizes the data as being generated by the addition of 4 underlying trends: a linearly increasing function; an approximately periodic function with a period of 1.0 years, and with linearly increasing amplitude; a

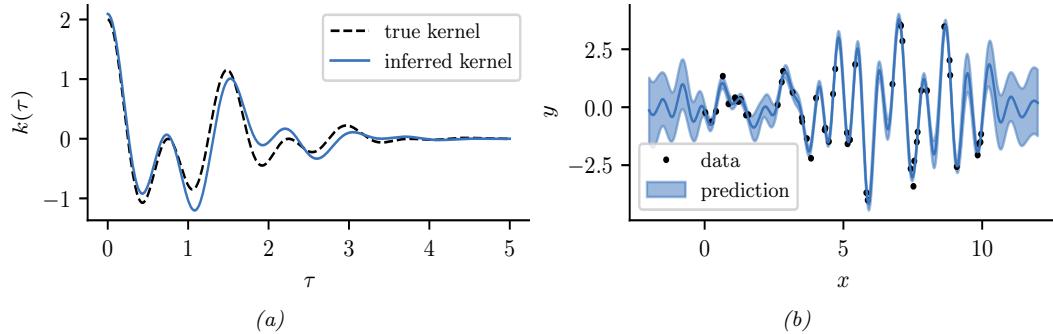


Figure 18.23: Illustration of a GP with a spectral mixture kernel in 1d. (a) Learned vs true kernel. (b) Predictions using learned kernel. Generated by [gp_spectral_mixture.ipynb](#).

smooth function; and uncorrelated noise with linearly increasing standard deviation.

Recently, [Sun+18] showed how to create a DNN which learns the kernel given two input vectors. The hidden units are defined as sums and products of elementary kernels, as in the above search based approach. However, the DNN can be trained in a differentiable way, so is much faster.

18.6.5 Spectral mixture kernel learning

Any shift-invariant (stationary) kernel can be converted via the Fourier transform to its dual form, known as its **spectral density**. This means that learning the spectral density is equivalent to learning any shift-invariant kernel. For example, if we take the Fourier transform of an RBF kernel, we get a Gaussian spectral density centered at the origin. If we take the Fourier transform of a Matérn kernel, we get a Student spectral density centred at the origin. Thus standard approaches to multiple kernel learning, which typically involve additive compositions of RBF and Matérn kernels with different length-scale parameters, amount to density estimation with a scale mixture of Gaussian or Student distributions at the origin. Such models are very inflexible for density estimation, and thus also very limited in being able to perform kernel learning.

On the other hand, *scale-location* mixture of Gaussians can model any density to arbitrary precision. Moreover, with even a small number of components these mixtures of Gaussians are highly flexible. Thus a spectral density corresponding to a scale-location mixture of Gaussians forms an expressive basis for all shift-invariant kernels. One can evaluate the inverse Fourier transform for a Gaussian mixture analytically, to derive the **spectral mixture kernel** [WA13], which we can express for one-dimensional inputs x as:

$$\mathcal{K}(x, x') = \sum_i w_i \cos((x - x')(2\pi\mu_i)) \exp(-2\pi^2(x - x')^2 v_i) \quad (18.172)$$

The mixture weights w_i , as well as the means μ_i and variances v_i of the Gaussians in the spectral density, can be learned by empirical Bayes optimization (Section 18.6.1) or in a fully-Bayesian procedure (Section 18.6.2) [Jan+17]. We illustrate the former approach in Figure 18.23.

By learning the parameters of the spectral mixture kernel, we can discover representations that enable extrapolation — to make reasonable predictions far away from the data. For example, in Sec-

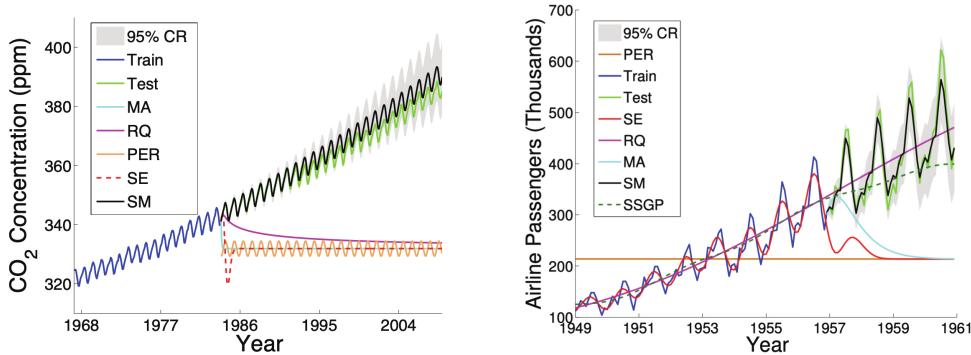


Figure 18.24: Extrapolations (point predictions and 95% credible set) on CO_2 and airline datasets using Gaussian processes with Matérn, rational quadratic, periodic, RBF (SE), and spectral mixture kernels, each with hyperparameters learned using empirical Bayes. From [Wil14].

tion 18.8.1, compositions of kernels are carefully hand-crafted to extrapolate CO_2 concentrations. But in this instance, the human statistician is doing all of the interesting representation learning. Figure Figure 18.24 shows Gaussian processes with learned spectral mixture kernels instead automatically extrapolating on CO_2 and airline passenger problems.

These kernels can also be used to extrapolate higher dimensional large-scale spatio-temporal patterns. Large datasets can provide relatively more information for expressive kernel learning. However, scaling an expressive kernel learning approach poses different challenges than scaling a standard Gaussian process model. One faces additional computational constraints, and the need to retain significant model structure for expressing the rich information available in a large dataset. Indeed, in Figure 18.24 we can separately understand the effects of the kernel learning approach and scalable inference procedure, in being able to discover structure necessary to extrapolate textures. An expressive kernel model and a scalable inference approach that preserves a *non-parametric* representation are needed for good performance.

Structure exploiting inference procedures, such as Kronecker methods, as well as KISS-GP and conjugate gradient based approaches, are appropriate for these tasks — since they generally preserve or exploit existing structure, rather than introducing approximations that corrupt the structure. Spectral mixture kernels combined with these scalable inference techniques have been used to great effect for spatiotemporal extrapolation problems, including land-surface temperature forecasting, epidemiological modeling, and policy-relevant applications.

18.6.6 Deep kernel learning

Deep kernel learning [SH07; Wil+16] combines the structural properties of neural networks with the non-parametric flexibility and uncertainty representation provided by Gaussian processes. For example, we can define a “deep RBF kernel” as follows:

$$\mathcal{K}_{\theta}(\mathbf{x}, \mathbf{x}') = \exp \left[-\frac{1}{2\sigma^2} \|\mathbf{h}_{\theta}^L(\mathbf{x}) - \mathbf{h}_{\theta}^L(\mathbf{x}')\|^2 \right] \quad (18.173)$$