Contents

1 Math and Machine Learning Basics **7**

1.1 Linear Algebra (Quick Review) (Ch. 2) 8

1.1.1 Example: Principal Component Analysis 10

1.2 Probability & Information Theory (Quick Review) (Ch. 3) 12

1.3 Numerical Computation (Ch. 4) 14

1.4 Machine Learning Basics (Ch. 5) 17

1.4.1 Estimators, Bias and Variance (5.4) 17

1.4.2 Maximum Likelihood Estimation (5.5) 19

1.4.3 Bayesian Statistics (5.6) 20

1.4.4 Supervised Learning Algorithms (5.7) 22

2 Deep Networks: Modern Practices **23**

2.1 Deep Feedforward Networks (Ch. 6) 24

2.1.1 Back-Propagation (6.5) 25

2.2 Regularization for Deep Learning (Ch. 7) 26

2.3 Optimization for Training Deep Models (Ch. 8) 28

2.4 Convolutional Neural Networks (Ch. 9) 32

2.5 Sequence Modeling (RNNs) (Ch. 10) 35

2.5.1 Review: The Basics of RNNs 35

2.5.2 RNNs as Directed Graphical Models 40

2.5.3 Challenge of Long-Term Deps. (10.7) 42

2.5.4 LSTMs and Other Gated RNNs (10.10) 43

2.6 Applications (Ch. 12) 44

2.6.1 Natural Language Processing (12.4) 44

2.6.2 Neural Language Models (12.4.2) 45

3 Deep Learning Research **46**

3.1 Linear Factor Models (Ch. 13) 47

3.2 Autoencoders (Ch. 14) 50
3.3 Representation Learning (Ch. 15) 51
3.4 Structured Probabilistic Models for DL (Ch. 16) 52
3.4.1 Sampling from Graphical Models 54
3.4.2 Inference and Approximate Inference 54
3.5 Monte Carlo Methods (Ch. 17) 56
3.6 Confronting the Partition Function (Ch. 18) 58
3.7 Approximate Inference (Ch. 19) 59
3.8 Deep Generative Models (Ch. 20) 61

4 Papers and Tutorials **66**
4.1 WaveNet 69
4.2 Neural Style 73
4.3 Neural Conversation Model 75
4.4 NMT By Jointly Learning to Align & Translate 77
4.4.1 Detailed Model Architecture 78
4.5 Effective Approaches to Attention-Based NMT 80
4.6 Using Large Vocabularies for NMT 82
4.7 Candidate Sampling – TensorFlow 85
4.8 Attention Terminology 87
4.9 TextRank 89
4.9.1 Keyword Extraction 91
4.9.2 Sentence Extraction 92
4.10 Simple Baseline for Sentence Embeddings 93
4.11 Survey of Text Clustering Algorithms 95
4.11.1 Feature Selection and Transformation Methods 95
4.11.2 Distance-based Clustering Algorithms 98
4.11.3 Probabilistic Document Clustering and Topic Models 99
4.11.4 Online Clustering with Text Streams 101
4.11.5 Semi-Supervised Clustering 102

4.12 Deep Sentence Embedding Using LSTMs 103
4.13 Clustering Massive Text Streams 106
4.14 Supervised Universal Sentence Representations (InferSent) 108
4.15 Dist. Rep. of Sentences from Unlabeled Data (FastSent) 109
4.16 Latent Dirichlet Allocation 111
4.17 Conditional Random Fields 114
4.18 Attention Is All You Need 117
4.19 Hierarchical Attention Networks 120
4.20 Joint Event Extraction via RNNs 123
4.21 Event Extraction via Bidi-LSTM Tensor NNs 125
4.22 Reasoning with Neural Tensor Networks 127
4.23 Language to Logical Form with Neural Attention 128
4.24 Seq2SQL: Generating Structured Queries from NL using RL 130
4.25 SLING: A Framework for Frame Semantic Parsing 133
4.26 Poincaré Embeddings for Learning Hierarchical Representations 135
4.27 Enriching Word Vectors with Subword Information (FastText) 137
4.28 DeepWalk: Online Learning of Social Representations 139
4.29 Review of Relational Machine Learning for Knowledge Graphs 141
4.30 Fast Top-K Search in Knowledge Graphs 144
4.31 Dynamic Recurrent Acyclic Graphical Neural Networks (DRAGNN) 146
4.31.1 More Detail: Arc-Standard Transition System 149
4.32 Neural Architecture Search with Reinforcement Learning 150
4.33 Joint Extraction of Events and Entities within a Document Context 152
4.34 Globally Normalized Transition-Based Neural Networks 155
4.35 An Introduction to Conditional Random Fields 158
4.35.1 Inference (Sec. 4) 162
4.35.2 Parameter Estimation (Sec. 5) 165
4.35.3 Related Work and Future Directions (Sec. 6) 168
4.36 Co-sampling: Training Robust Networks for Extremely Noisy Supervision . . . 169

4.37 Hidden-Unit Conditional Random Fields 170
4.37.1 Detailed Derivations 172
4.38 Pre-training of Hidden-Unit CRFs 177
4.39 Structured Attention Networks 179
4.40 Neural Conditional Random Fields 181
4.41 Bidirectional LSTM-CRF Models for Sequence Tagging 183
4.42 Relation Extraction: A Survey 184
4.43 Neural Relation Extraction with Selective Attention over Instances 187
4.44 On Herding and the Perceptron Cycling Theorem 189
4.45 Non-Convex Optimization for Machine Learning 191
4.45.1 Non-Convex Projected Gradient Descent (3) 194
4.46 Improving Language Understanding by Generative Pre-Training 195
4.47 Deep Contextualized Word Representations 196
4.48 Exploring the Limits of Language Modeling 198

5 NLP with Deep Learning 200
5.1 Word Vector Representations (Lec 2) 201
5.2 GloVe (Lec 3) 204

6 Speech and Language Processing 207
6.1 Introduction (Ch. 1 2nd Ed.) 208
6.2 Morphology (Ch. 3 2nd Ed.) 209
6.3 N-Grams (Ch. 6 2nd Ed.) 210
6.4 Naive Bayes and Sentiment (Ch. 6 3rd Ed.) 212
6.5 Hidden Markov Models (Ch. 9 3rd Ed.) 214
6.6 POS Tagging (Ch. 10 3rd Ed.) 217
6.7 Formal Grammars (Ch. 11 3rd Ed.) 220
6.8 Vector Semantics (Ch. 15) 221
6.9 Semantics with Dense Vectors (Ch. 16) 224
6.10 Information Extraction (Ch. 21 3rd Ed) 227

7 Probabilistic Graphical Models **230**

7.1 Foundations (Ch. 2) 231

7.1.1 Appendix 233

7.1.2 L-BFGS 236

7.1.3 Exercises 239

7.2 The Bayesian Network Representation (Ch. 3) 241

7.3 Undirected Graphical Models (Ch. 4) 245

7.3.1 Exercises 252

7.4 Local Probabilistic Models (Ch. 5) 254

7.5 Template-Based Representations (Ch. 6) 255

7.6 Gaussian Network Models (Ch. 7) 258

7.7 Variable Elimination (Ch. 9) 259

7.8 Clique Trees (Ch. 10) 263

7.9 Inference as Optimization (Ch. 11) 267

7.10 Parameter Estimation (Ch. 17) 268

7.11 Partially Observed Data (Ch. 19) 270

8 Information Theory, Inference, and Learning Algorithms **272**

8.1 Introduction to Information Theory (Ch. 1) 273

8.2 Probability, Entropy, and Inference (Ch. 2) 275

8.2.1 More About Inference (Ch. 3 Summary) 276

8.3 The Source Coding Theorem (Ch. 4) 278

8.3.1 Data Compression and Typicality 280

8.3.2 Further Analysis and Q&A 282

8.4 Monte Carlo Methods (Ch. 29) 284

8.5 Variational Methods (Ch. 33) 286

9 Machine Learning: A Probabilistic Perspective **288**

9.1 Probability (Ch. 2) 289

9.2 Generative Models for Discrete Data (Ch. 3) 291

9.2.1 Exercises . . . 295
9.3 Gaussian Models (Ch. 4) . . . 296
9.4 Linear Regression (Ch. 7) . . . 300
9.5 Generalized Linear Models and the Exponential Family (Ch. 9) . . . 303
9.6 Mixture Models and the EM Algorithm (Ch. 11) . . . 306
9.7 Markov and Hidden Markov Models (Ch. 17) . . . 307
9.8 Undirected Graphical Models (Ch. 19) . . . 308

10 Convex Optimization **309**
10.1 Convex Sets (Ch. 2) . . . 310

11 Bayesian Data Analysis **312**
11.1 Probability and Inference (Ch. 1) . . . 313
11.2 Single-Parameter Models (Ch. 2) . . . 315
11.3 Asymptotics and Connections to Non-Bayesian Approaches (Ch. 4) . . . 318
11.4 Gaussian Process Models (Ch. 21) . . . 321

12 Gaussian Processes for Machine Learning **323**
12.1 Regression (Ch. 2) . . . 324

13 Blogs **327**
13.1 Conv Nets: A Modular Perspective . . . 328
13.2 Understanding Convolutions . . . 329
13.3 Deep Reinforcement Learning . . . 331
13.4 Deep Learning for Chatbots (WildML) . . . 333
13.5 Attentional Interfaces – Neural Perspective . . . 335

14 Appendix **336**
14.1 Common Distributions and Models . . . 337
14.2 Math . . . 338
14.3 Main Tasks in NLP . . . 341
14.4 Misc. Topics . . . 344

Math and Machine Learning Basics

Contents

1.1 Linear Algebra (Quick Review) (Ch. 2) 8

1.1.1 Example: Principal Component Analysis 10

1.2 Probability & Information Theory (Quick Review) (Ch. 3) 12

1.3 Numerical Computation (Ch. 4) 14

1.4 Machine Learning Basics (Ch. 5) 17

1.4.1 Estimators, Bias and Variance (5.4) 17

1.4.2 Maximum Likelihood Estimation (5.5) 19

1.4.3 Bayesian Statistics (5.6) 20

1.4.4 Supervised Learning Algorithms (5.7) 22

- For A^{-1} to exist, $Ax = b$ must have exactly one solution for every value of b. Determining whether a solution exists $\forall b \in \mathbb{R}^m$ means requiring that the **column space** (range) of A be all of $\mathbb{R}^m$. It is helpful to see Ax expanded out explicitly in this way:

Unless stated otherwise, assume $A \in \mathbb{R}^{m \times n}$

$$\boldsymbol{A}\boldsymbol{x} = \sum_i x_i \boldsymbol{A}_{:,i} = x_1 \begin{pmatrix} A_{1,1} \\ \vdots \\ A_{m,1} \end{pmatrix} + \cdots + x_m \begin{pmatrix} A_{1,n} \\ \vdots \\ A_{m,n} \end{pmatrix} \tag{2.27}$$

  → Necessary: $\boldsymbol{A}$ must have at least m columns ($n \geq m$). ("wide").
  → Necessary *and* sufficient: matrix must contain at least one set of m linearly independent columns.
  → Invertibility: In addition to above, need matrix to be *square* (re: at most m columns $\wedge$ at least m columns).

- A square matrix with linearly dependent columns is known as **singular**. A (necessarily square) matrix is singular if and only if one or more eigenvalues are zero.

- A **norm** is any function f that satisfies the following properties:

$||x||_\infty = \max_i |x_i|$

$$f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0} \tag{1}$$
$$f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y}) \tag{2}$$
$$\forall \alpha \in \mathbb{R},\ f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x}) \tag{3}$$

- An **orthogonal matrix** is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

Note that orthonorm cols implies orthonorm rows (if square). To prove, consider the relationship between $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^T$

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} \tag{2.37}$$
$$\boldsymbol{A}^{-1} = \boldsymbol{A}^T \tag{2.38}$$

- Suppose square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has n linearly independent eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$. The **eigendecomposition** of $\boldsymbol{A}$ is then given by[1]

$$\boldsymbol{A} = \boldsymbol{V}\,\mathrm{diag}\,(\boldsymbol{\lambda})\ \boldsymbol{V}^{-1} \tag{2.40}$$

  In the special case where $\boldsymbol{A}$ is real-symmetric, $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$. **Interpretation:** $\boldsymbol{A}\boldsymbol{x}$ can be decomposed into the following three steps:

All real-symmetric $\boldsymbol{A}$ have an eigendecomposition, **but it might not be unique!**

[1] This appear to imply that unless the columns of V are also normalized, can't guarantee that its inverse equals its transpose? (since that is the only difference between it and an orthogonal matrix)

1) **Change of basis**: The vector $(\boldsymbol{Q}^T\boldsymbol{x})$ can be thought of as how $\boldsymbol{x}$ would appear in the basis of eigenvectors of $\boldsymbol{A}$.
2) **Scale**: Next, we scale each component $(\boldsymbol{Q}^T\boldsymbol{x})_i$ by an amount λ_i, yielding the new vector $(\boldsymbol{\Lambda}(\boldsymbol{Q}^T\boldsymbol{x}))$.
3) **Change of basis**: Finally, we rotate this new vector back from the eigen-basis into its original basis, yielding the transformed result of $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{x}$.

A common convention to sort the entries of Λ in descending order.

- **Positive definite**: all λ are positive; **positive semidefinite**: all λ are positive or zero.

  $\rightarrow$ PSD: $\forall \boldsymbol{x}, \quad \boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} \geq 0$
  $\rightarrow$ PD: $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = 0 \;\Rightarrow\; \boldsymbol{x} = \boldsymbol{0}$.[2]

- Any real matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ has a **singular value decomposition** of the form,

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T \tag{10}$$

$$\boldsymbol{U} \in \mathbb{R}^{m\times m} \tag{7}$$
$$\boldsymbol{D} \in \mathbb{R}^{m\times n} \tag{8}$$
$$\boldsymbol{V} \in \mathbb{R}^{n\times n} \tag{9}$$

  where both $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices, and $\boldsymbol{D}$ is diagonal.

  - The **singular values** are the diagonal entries $\boldsymbol{D}_{ii}$.
  - The **left(right)-singular vectors** are the columns of $\boldsymbol{U}(\boldsymbol{V})$.
  - Eigenvectors of $\boldsymbol{A}\boldsymbol{A}^T$ are the L-S vectors. Eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ are the R-S vectors. The eigenvalues of both $\boldsymbol{A}\boldsymbol{A}^T$ and $\boldsymbol{A}^T\boldsymbol{A}$ are given by the singular values squared.

- The Moore-Penrose **pseudoinverse**, denoted $\boldsymbol{A}^+$, enables us to find an "inverse" of sorts for a (possibly) non-square matrix $\boldsymbol{A}$. Most algorithms compute $\boldsymbol{A}^+$ via

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^T \tag{11}$$

$\boldsymbol{A}^+$ is useful, e.g., when we want to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ by left-multiplying each side to obtain $\boldsymbol{x} = \boldsymbol{B}\boldsymbol{y}$. It is far more likely for solution(s) to exist when $\boldsymbol{A}$ is wider than it is tall.

- The **determinant** of a matrix is $\det(\boldsymbol{A}) = \prod_i \lambda_i$. Conceptually, $|\det(\boldsymbol{A})|$ tells how much [multiplication by] $\boldsymbol{A}$ expands/contracts space. If $\det(\boldsymbol{A}) = 1$, the transformation preserves volume.

[2] I proved this and it made me happy inside. Check it out. Let $\boldsymbol{A}$ be positive definite. Then

$$\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{x} \tag{4}$$
$$= \sum_i (\boldsymbol{Q}^T\boldsymbol{x})_i \lambda_i (\boldsymbol{Q}^T\boldsymbol{x})_i \tag{5}$$
$$= \sum_i \lambda_i (\boldsymbol{Q}^T\boldsymbol{x})_i^2 \tag{6}$$

Since all terms in the summation are non-negative and all $\lambda_i > 0$, we have that $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = 0$ if and only if $(\boldsymbol{Q}^T\boldsymbol{x})_i = 0 = \boldsymbol{q}^{(i)} \cdot \boldsymbol{x}$ for all i. Since the set of eigenvectors $\{\boldsymbol{q}^{(i)}\}$ form an orthonormal basis, we have that $\boldsymbol{x}$ must be the zero vector.

Example: Principal Component Analysis

Task. Say we want to apply lossy compression (less memory, but may lose precision) to a collection of m points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$. We will do this by converting each $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ to some $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$ $(l < n)$, i.e. finding functions f and g such that:

$$f(\boldsymbol{x}) = \boldsymbol{c} \quad \text{and} \quad \boldsymbol{x} \approx g(f(\boldsymbol{x})) \tag{12}$$

Decoding function (g). As is, we still have a rather general task to solve. *PCA* is defined by choosing $g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, with $\boldsymbol{D} \in \mathbb{R}^{n \times l}$, where all columns of $\boldsymbol{D}$ are both (1) orthogonal and (2) unit norm.

Encoding function (f). Now we need a way of mapping $\boldsymbol{x}$ to $\boldsymbol{c}$ such that $g(\boldsymbol{c})$ will give us back a vector optimally close to $\boldsymbol{x}$. We've already defined g, so this amount to finding the optimal $\boldsymbol{c}*$ such that:

$$\boldsymbol{c}* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - g(\boldsymbol{c})||_2^2 \tag{13}$$

$$(\boldsymbol{x} - g(\boldsymbol{c}))^T(\boldsymbol{x} - g(\boldsymbol{c})) = \boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{x}^T g(\boldsymbol{c}) + g(\boldsymbol{c})^T g(\boldsymbol{c}) \tag{14}$$

$$\boldsymbol{c}* = \arg\min_{\boldsymbol{c}} \left[-2\boldsymbol{x}^T\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^T\boldsymbol{c}\right] \tag{15}$$

$$= \boldsymbol{D}^T\boldsymbol{x} = f(\boldsymbol{x}) \tag{16}$$

which means the PCA *reconstruction operation* is defined as $r(\boldsymbol{x}) = \boldsymbol{D}\boldsymbol{D}^T\boldsymbol{x}$.

Optimal $\boldsymbol{D}$. It is important to notice that we've been able to determine e.g. the optimal $\boldsymbol{c}*$ *for some* $\boldsymbol{x}$ because each $\boldsymbol{x}$ has a (allowably) different $\boldsymbol{c}*$. However, we use *the same* matrix $\boldsymbol{D}$ for all our samples $\boldsymbol{x}^{(i)}$, and thus must optimize it over all points in our collection. With that out of the way, we just do what we always do: minimize over the L^2 distance between points and their reconstruction. Formally, we minimize the Frobenius norm of the matrix of errors:

$$\boldsymbol{D}* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} \left(\boldsymbol{x}_j^{(i)} - r(\boldsymbol{x}^{(i)})_j\right)^2} \quad s.t. \quad \boldsymbol{D}^T\boldsymbol{D} = \boldsymbol{I} \tag{17}$$

Consider the case of $l = 1$ which means $\boldsymbol{D} = \boldsymbol{d} \in \mathbb{R}^n$. In this case, after [insert math here], we obtain

$$\boldsymbol{d}* = \arg\max_{\boldsymbol{d}} Tr\left(\boldsymbol{d}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{d}\right) \quad s.t. \quad \boldsymbol{d}^T \boldsymbol{d} = 1 \tag{18}$$

where, as usual, $\boldsymbol{X} \in \mathbb{R}^{m,n}$. It should be clear that the optimal $\boldsymbol{d}$ is just the largest eigenvector of $\boldsymbol{X}^T \boldsymbol{X}$.

Probability & Information Theory (Quick Review) (Ch. 3)

Expectation. For some function $f(x)$, $\mathbb{E}_{x\sim P}[f(x)]$ is the mean value that f takes on when x is drawn from P. The formula for discrete and continuous variables, respectively is as follows:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x) \tag{3.9}$$

$$\mathbb{E}_{x\sim P}[f(x)] = \int p(x)f(x)\mathrm{d}x \tag{3.10}$$

Variance. A measure of how much the values of a function of a random variable x vary as we sample different values of x from its distribution.

$$\mathrm{Var}[f(x)] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{3.11}$$

Covariance. Gives some sense of how much two values are *linearly* related to each other, as well as the *scale* of these variables.

$$\mathrm{Cov}[f(x),\ g(x)] = \mathbb{E}[\ \ (f(x) - \mathbb{E}[f(x)])\ \ (g(x) - \mathbb{E}[g(x)])\ \ ] \tag{3.13}$$

→ Large $|\mathrm{Cov}[f,\ g]|$ means the function values change a lot and both functions are far from their means at the same time.

→ **Correlation** normalizes the contribution of each variable in order to measure only how much the variables are related.

Covariance Matrix of a random vector $\mathbf{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that

$$\mathrm{Cov}[\mathbf{x}]_{i,j} = \mathrm{Cov}[x_i,\ x_j] \tag{3.14}$$

and if we want the "sample" covariance matrix taken over m data point samples, then

$$\Sigma := \frac{1}{m}\sum_{k=1}^{m}(x_k - \bar{x})(x_k - \bar{x})^T \tag{19}$$

where m is the number of data points.

Measure Theory.

- A set of points that is negligibly small is said to have **measure zero**. In practical terms, think of such a set as occupying no volume in the space we are measuring (interested in).

In $\mathbb{R}^2$, a line has measure zero.

- A property that holds **almost everywhere** holds throughout all space except for on a set of measure zero.

Functions of RVs.

- **Common mistake:** Suppose $\mathbf{y} = g(\mathbf{x})$, and g is invertible/continuous/differentiable. It is NOT true that $p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y}))$. This fails to account for the distortion of [probability] space introduced by g. Rather,

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x}))\left|\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right| \tag{3.47}$$

Information Theory. Denote the **self-information** of an event $\mathbf{x} = x$ to be

$$I(x) \triangleq -\log P(x) \tag{20}$$

where log is always assumed to be the natural logarithm. We can quantify the amount of uncertainty in an entire probability distribution using the **Shannon entropy**,

$$H(\mathbf{x}) = \mathbb{E}_{\mathbf{x}\sim P}\left[I(x)\right] = -\mathbb{E}_{\mathbf{x}\sim P}\left[\log P(x)\right] \tag{21}$$

which gives the expected amount of information in an event drawn from that distribution. Taking it a step further, say we have two separate probability distributions $P(\mathbf{x})$ and $Q(\mathbf{x})$. We can measure how different these distributions are with the **Kullback-Leibler (KL) divergence**:

$$D_{KL}\left(P||Q\right) \triangleq \mathbb{E}_{\mathbf{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathbf{x}\sim P}\left[\log P(x) - \log Q(x)\right] \tag{22}$$

Note that the expectation is taken over P, thus making D_{KL} not symmetric (and thus not a true distance measure), since $D_{KL}(P||Q) \neq D_{KL}(Q||P)$. Finally, a closely related quantity is the **cross-entropy**, $H(P, Q)$, defined as:

$$\begin{aligned} H(P,Q) &\triangleq H(P) + D_{KL}(P||Q) && (23)\\ &= -\mathbb{E}_{\mathbf{x}\sim P}\left[\log Q(x)\right] && (24) \end{aligned}$$

Some terminology. **Underflow** is when numbers near zero are rounded to zero. Similarly, **overflow** is when large [magnitude] numbers are approximated as $\pm\infty$. **Conditioning** refers to how rapidly a function changes w.r.t. small changes in its inputs. Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A}$ has an eigenvalue decomposition, its *condition number* is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right| \tag{4.2}$$

which is the ratio of the magnitude of the largest and smallest eigenvalue. When this is large, matrix inversion is sensitive to error in the input [of f(x)].

Gradient-based optimization. Recall from basic calculus that the **directional derivative** of $f(\boldsymbol{x})$ in direction $\hat{\boldsymbol{u}}$ (a unit vector) is defined as the slope of the function f in direction $\hat{\boldsymbol{u}}$. By definition of the derivative, this is given by (with $\boldsymbol{v} := \boldsymbol{x} + \alpha\hat{\boldsymbol{u}}$)

$$\begin{aligned}
\lim_{\alpha\to 0} \frac{f(\boldsymbol{x}+\alpha\hat{\boldsymbol{u}}) - f(\boldsymbol{x})}{\alpha} &= \left.\frac{\partial f(\boldsymbol{x}+\alpha\hat{\boldsymbol{u}})}{\partial\alpha}\right|_{\alpha=0} && (25)\\
&= \sum_i \left.\frac{\partial f(\boldsymbol{v})}{\partial v_i}\frac{\partial v_i}{\partial\alpha}\right|_{\alpha=0} && (26)\\
&= \sum_i \left.(\nabla_{\boldsymbol{v}} f(\boldsymbol{v}))_i \; u_i\right|_{\alpha=0} && (27)\\
&= \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{v}} f(\boldsymbol{v})\Big|_{\alpha=0} && (28)\\
&= \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) && (29)
\end{aligned}$$

where it's important to recognize the distinction between $\lim_{\alpha\to 0}$ and *setting* α to zero, which is denoted by $|_{\alpha=0}$. If we want to *find* the direction $\hat{\boldsymbol{u}}$ such that this directional derivative is a minimum, i.e.

$$\begin{aligned}
\hat{\boldsymbol{u}}^* &= \underset{\hat{\boldsymbol{u}},\hat{\boldsymbol{u}}^T\hat{\boldsymbol{u}}=1}{\arg\min}\; \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) && (30)\\
&= \underset{\hat{\boldsymbol{u}},\hat{\boldsymbol{u}}^T\hat{\boldsymbol{u}}=1}{\arg\min}\; ||\hat{\boldsymbol{u}}||_2 \; ||\nabla_{\boldsymbol{x}} f(\boldsymbol{x})||_2 \; \cos(\boldsymbol{\theta}) && (31)\\
&= \cos(\boldsymbol{\theta}) && (32)
\end{aligned}$$

and we see that $\hat{\boldsymbol{u}}$ points in the opposite direction as the gradient.

Jacobian and Hessian Matrices. For when we want partial derivatives of some function f whose input and output are both vectors. The **Jacobian matrix** contains all such partial derivatives. Sometimes we want to know about second derivatives too, since this tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone. The **Hessian matrix** $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

$f : \mathbb{R}^m \to \mathbb{R}^n$
$\boldsymbol{J} \in \mathbb{R}^{n \times m}$ where
$J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$

The Hessian is the Jacobian of the gradient.

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) \tag{4.6}$$

The second derivative in a specific direction $\hat{\boldsymbol{d}}$ is given by $\hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}}$[3]. It tells us how well we can expect a gradient descent step to perform. How so? Well, it shows up in the second-order approximation to the function $f(\boldsymbol{x})$ about our current spot, which we can denote $\boldsymbol{x}^{(0)}$. The standard gradient descent step will move us from $\boldsymbol{x}^{(0)} \to \boldsymbol{x}^{(0)} - \epsilon g$, where g is the gradient evaluated at $\boldsymbol{x}^{(0)}$. Plugging this in to the 2nd order approximation shows us how $\boldsymbol{H}$ can give information related to how "good" of a step that really was. Mathematically,

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^T \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)}) \tag{4.8}$$

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} \tag{4.9}$$

If $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ is positive, then we can easily solve for the optimal $\epsilon = \epsilon^*$ that decreases the Taylor series approximation as

$$\epsilon^* = \frac{\boldsymbol{g}^T \boldsymbol{g}}{\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}} \tag{4.10}$$

which can be as low as $1/\lambda_{max}$ (the worst case), and as high as $1/\lambda_{min}$ with the λ being the eigenvalues of the Hessian. The best (and perhaps only) way to take what we learned about the "second derivative test" in single-variable calculus and apply it to the multidimensional case with $\boldsymbol{H}$ is by using the *eigendecomposition of* $\boldsymbol{H}$. Why? Because we can examine the eigen<u>values</u> of the Hessian to determine whether the critical point $\boldsymbol{x}^{(0)}$ is a local maximum, local minimum, or saddle point[4]. If all eigenvalues are positive (remember that this is equivalent to saying that the Hessian is **positive definite**!), the point is a local minimum.

The condition number of the Hessian at a given point can give us an idea about how much the second derivatives (along different directions) differ from each other

[3] In the same manner that I derived equation 29, we can derive the second derivative in a specified direction $\hat{\boldsymbol{d}}$:

$$\frac{\partial^2}{\partial \alpha^2} f(\boldsymbol{x} + \alpha \hat{\boldsymbol{d}})\big|_{\alpha=0} = \frac{\partial}{\partial \alpha} \hat{\boldsymbol{d}}^T \nabla_{\boldsymbol{v}} f(\boldsymbol{v})\big|_{\alpha=0} \tag{33}$$

$$= \sum_i d_i \frac{\partial}{\partial \alpha} \frac{\partial f(\boldsymbol{v})}{\partial v_i} \Big|_{\alpha=0} \tag{34}$$

$$= \sum_i d_i \frac{\partial}{\partial v_i} \frac{\partial f(\boldsymbol{v})}{\partial \alpha} \Big|_{\alpha=0} \tag{35}$$

$$= \sum_i \sum_j d_i \frac{\partial^2 f(\boldsymbol{v})}{\partial v_i v_j} d_j \Big|_{\alpha=0} \tag{36}$$

$$= \hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}} \tag{37}$$

[4] Emphasis on "values" in "eigen<u>values</u>" because it's important not to get tripped up here about what the

Constrained optimization: minimizing/maximizing a function $f(\boldsymbol{x})$ constrained to only values of $\boldsymbol{x}$ in some set $\mathbb{S}$. One way of approaching such a problem is to re-design the unconstrained optimization problem such that the re-designed problem's solution satisfies the constraints. For example, to minimize $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^2$ with constraint $||\boldsymbol{x}||_2 = 1$, we can minimize $g(\theta) = f([\cos\theta, \sin\theta]^T)$ wrt θ, then return $[\cos\theta, \sin\theta]^T$ as the solution to the original problem.

The **Karush-Kuhn-Tucker** (KKT) approach, a generalization of **Lagrange multipliers**, provides a general approach for re-designing the optimization problem, with procedure as follows:

1. Find m functions $g^{(i)}$ and n functions $h^{(j)}$ such that your set of allowed values $\mathbb{S}$ can be written

$$\mathbb{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\} \tag{38}$$

   The equations involving $g^{(i)}$ are called the **equality constraints** and the inequalities involving $h^{(j)}$ are called the **inequality constraints**.
2. Introduce new variables λ_i (for the equality constraints) and α_j (for the inequality constraints). These are called the KKT multipliers. The generalized Lagrangian is then defined as

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}) \tag{39}$$

3. Solve the re-designed unconstrained optimization problem:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \tag{40}$$

   which has the same optimal objective function value and set of optimal points $\boldsymbol{x}$ as the original constrained problem, $\min_{\boldsymbol{x} \in \mathbb{S}} f(\boldsymbol{x})$. Any time the constraints are satisfied, the expression $\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ evaluates to $f(\boldsymbol{x})$, and any time a constraint is violated, the same expression evaluates to ∞.

eigenvectors of the Hessian mean. The reason for the decomposition is that it gives us an orthonormal basis (out of which we can get any direction) and therefore the magnitude of the second derivative along each of these directions as the eigenvalues.

Capacity, Overfitting, and Underfitting. Difference between ML and optimization is that, in addition to wanting low training error, we want **generalization error** (test error) to be low as well. The ideal model is an oracle that simply knows the true probability distribution $p(\boldsymbol{x}, y)$ that generates the data. The error incurred by such an oracle, due things like inherently stochastic mappings from $\boldsymbol{x}$ to y or other variables, is called the **Bayes error**. The **no free lunch theorem** states that, averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. Therefore, the goal of ML research is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of ML algorithms perform well on data drawn from the relevant data-generating distributions.

Estimators, Bias and Variance (5.4)

Point Estimation: attempt to provide "best" prediction of some quantity, such as some parameter or even a whole function. Formally, a point estimator or *statistic* is any function of the data:

$$\hat{\theta}_m = g\left(x^{(1)}, \ldots, x^{(m)}\right) \tag{5.19}$$

where, since the data is drawn from a random process, $\hat{\theta}$ is a random variable. **Function estimation** is identical in form, where we want to estimate some $f(x)$ with $\hat{f}$, a point estimator in *function space*.

Bias. Defined below, where the expectation is taken over the data-generating distribution[5]. Bias measures the expected deviation from the true value of the func/param.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{5.20}$$

TODO: Figure out how to derive $\mathbb{E}\left[\hat{\theta}_m^2\right]$ for Gaussian distribution [helpful link].

[5] May want to double-check this, but I'm fairly certain this is what the book meant when it said "data," based on later examples.

Bias-Variance Tradeoff.

→ **Conceptual Info**. Two sources of error for an estimator are (1) bias and (2) variance, which are both defined as deviations from a certain value. Bias gives deviation from the *true* value, while variance gives the [expected] deviation from this *expected* value.

→ **Summary of main formulas**.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{41}$$

$$\text{Var}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\left(\hat{\theta}_m - \mathbb{E}\left[\hat{\theta}_m\right]\right)^2\right] \tag{42}$$

→ **MSE decomposition**. The MSE of the estimates is given by[6]

$$\text{MSE} = \mathbb{E}\left[(\hat{\theta}_m - \theta)^2\right] \tag{5.53}$$

$$= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] \tag{5.54}$$

and desirable estimators are those with low MSE.

Consistency. As the number of training data points increases, we want the estimators to converge to the true values. Specifically, below are the definitions for *weak* and *strong* consistency, respectively.

$$\begin{aligned} \text{plim}_{m\to\infty}\hat{\theta}_m &= 0 \\ p\left(\lim_{m\to\infty}\hat{\theta}_m = \theta\right) &= 1 \end{aligned} \tag{5.55}$$

where the symbol plim means $P(|\hat{\theta}_m - \theta| > \epsilon) \to 0$ as $m \to \infty$.

[6] Derivation:

$$\text{MSE} = \mathbb{E}\left[\hat{\theta}^2 + \theta^2 - 2\theta\hat{\theta}\right] \tag{43}$$

$$= \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] \tag{44}$$

$$= (\mathbb{E}\left[\hat{\theta}\right]^2 - \mathbb{E}\left[\hat{\theta}\right]^2) + \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] \tag{45}$$

$$= \left(\mathbb{E}\left[\hat{\theta}\right]^2 + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right]\right) + \left(\mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2\right) \tag{46}$$

$$= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] \tag{47}$$

Maximum Likelihood Estimation (5.5)

Consider set of m examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn independently from the true (but unknown) $p_{data}(\mathbf{x})$. Let $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ be parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$. The maximum likelihood estimator for $\boldsymbol{\theta}$ can be expressed as

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} \left[\log p_{model}(\boldsymbol{x}; \boldsymbol{\theta})\right] \tag{5.59}$$

where we've chosen to express with log for underflow/gradient reasons. One interpretation of ML is to view it as minimizing the dissimilarity, as measured by the KL divergence[7], between $\hat{p}_{data}$ and p_{model}.

> Any loss consisting of a negative log-likelihood is a **cross-entropy** between the $\hat{p}_{data}$ distribution and the p_{model} distribution.

Thoughts: Let's look at D_{KL} in some more detail. First, I'll rewrite it with the explicit definition of $\mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log\left(\hat{p}_{data}(\boldsymbol{x})\right)\right]$:

$$\begin{aligned} D_{KL}(\hat{p}_{data} || p_{model}) &= \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log\left(\hat{p}_{data}(\boldsymbol{x})\right) - \log\left(p_{model}(\boldsymbol{x})\right)\right] && (48) \\ &= \left(\frac{1}{N}\left(\sum_{i=1}^{N} \log\left(\text{Counts}(\boldsymbol{x}_i)\right)\right) - \log N\right) - \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log\left(p_{model}(\boldsymbol{x})\right)\right] && (49) \end{aligned}$$

Note also that our goal is to find parameters $\boldsymbol{\theta}$ such that D_{KL} is minimized. It is for *this* reason, that we wish to optimize over $\boldsymbol{\theta}$, that minimizing D_{KL} amounts to maximizing the quantity, $\mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log\left(p_{model}(\boldsymbol{x})\right)\right]$. Sure, I can agree this is *true*, but **why is our goal to minimize D_{KL}, as opposed to minimizing $\mid D_{KL} \mid$?** I'm assuming it is because optimizing w.r.t. an absolute value is challenging numerically.

Conditional Log-Likelihood and MSE. We can readily generalize $\boldsymbol{\theta}_{ML}$ to estimate a conditional probability $p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ in order to predict $\mathbf{y}$ given $\mathbf{x}$, since

We are assuming the examples are i.i.d. here.

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.63}$$

where $\boldsymbol{x}^{(i)}$ are fed as *inputs* to the model; this is why we can formulate MLE as a conditional probability.

[7] The KL divergence is given by

$$D_{KL}(\hat{p}_{data} || p_{model}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} \left[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})\right] \tag{5.60}$$

Bayesian Statistics (5.6)

Distinction between frequentist and bayesian approach:

- **Frequentist:** Estimate $\boldsymbol{\theta} \longrightarrow$ make predictions thereafter based on this estimate.
- **Bayesian:** Consider all possible values of $\boldsymbol{\theta}$ when making predictions.

The prior. Before observing the data, we represent our knowledge of $\boldsymbol{\theta}$ using the **prior probability distribution** $p(\boldsymbol{\theta})$. Unlike maximum likelihood, which makes predictions using a *point estimate* of $\boldsymbol{\theta}$ (a single value), the Bayesian approach uses Bayes' rule to make predictions using the *full distribution* over $\boldsymbol{\theta}$. In other words, rather than focusing on the most accurate value estimate of $\boldsymbol{\theta}$, we instead focus on pinning down a range of possible $\boldsymbol{\theta}$ values and how likely we believe each of these values to be.

It is common to choose a high-entropy prior, e.g. uniform.

So what happens to $\boldsymbol{\theta}$ after we observe the data? We update it using Bayes' rule[8]:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})} \tag{50}$$

Note that we still haven't mentioned how to actually make *predictions*. Since we no longer have just one value for $\boldsymbol{\theta}$, but rather we have a posterior *distribution* $p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})$, we must integrate over this to get the predicted likelihood of the next sample $x^{(m+1)}$:

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) \mathrm{d}\boldsymbol{\theta} \tag{51}$$

$$= \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})} \left[p(x^{(m+1)} \mid \boldsymbol{\theta}) \right] \tag{52}$$

Linear Regression: MLE vs. Bayesian. Both want to model the conditional distribution $p(y \mid \boldsymbol{x})$ (the conditional likelihood). To derive the standard linear regression algorithm, we *define*

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y;\ \hat{y}(\boldsymbol{x}; \boldsymbol{w}),\ \sigma^2\right) \tag{53}$$

$$\hat{y}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x} \tag{54}$$

Assume σ^2 is some fixed constant chosen by the user.

[8] In practice, we typically compute the denominator by simply normalizing the probability distribution, i.e. it is effectively the partition function.

- **Maximum Likelihood Approach**: We can use the definition above (and the i.i.d. assumption) to evaluate the conditional log-likelihood as

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{||\hat{y}^{(i)} - y^{(i)}||^2}{2\sigma^2} \tag{5.65}$$

  where only the last term has any dependence on $\boldsymbol{w}$. Therefore, to obtain $\boldsymbol{w}_{ML}$ we take the derivative of the last term w.r.t. $\boldsymbol{w}$, set that to zero, and solve for $\boldsymbol{w}$. We see that finding the $\boldsymbol{w}$ that maximizes the conditional log-likelihood is equivalent to finding the $\boldsymbol{w}$ that minimizes the training MSE.

  Recall that the training MSE is $\frac{1}{m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2$

- **Bayesian Approach**: Our conditional likelihood is already given in equation 53. Next, we must define a prior distribution over $\boldsymbol{w}$. As is common, we choose a Gaussian prior to express our high degree of uncertainty about $\boldsymbol{\theta}$ (implying we'll choose a relatively large variance):

$$p(\boldsymbol{w}) := \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \tag{55}$$

  Typically assume $\boldsymbol{\Lambda}_0 = \text{diag}\,(\boldsymbol{\lambda}_0)$

  We can then compute [the unnormalized] $p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w})$ [and then normalize it].

Maximum A Posteriori (MAP) Estimation. Often we either prefer a point estimate for θ, or we find out that computing the posterior distribution is intractable and a point estimate offers a tractable estimation. The obvious way of obtaining this while still taking the Bayesian route is to just argmax the posterior and use that as your point estimate:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p\left(\boldsymbol{x} \mid \boldsymbol{\theta}\right) + \log p(\boldsymbol{\theta}) \tag{56}$$

where the second form shows how this is basically maximum likelihood with incorporation of the prior. We don't want just any $\boldsymbol{\theta}$ that maximizes the likelihood of our data if there is virtually no chance of that value of $\boldsymbol{\theta}$ in the first place.

Supervised Learning Algorithms (5.7)

Logistic Regression. We've already seen that linear regression corresponds to the family

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y;\ \boldsymbol{\theta}^T\boldsymbol{x},\ \boldsymbol{I}\right) \tag{5.80}$$

which we can generalize to the binary **classification** scenario by interpreting as the probability of class 1. One way of doing this while ensuring the output is between 0 and 1 is to use the logistic sigmoid function:

$$p(y = 1 \ \mid \ \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T\boldsymbol{x}) \tag{5.81}$$

Equation 5.81 is the definition of logistic regression

Unfortunately, there is no closed-form solution for $\boldsymbol{\theta}$, so we must search via maximizing the log-likelihood.

Support Vector Machines. Driving by a linear function $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$ like logistic regression, but instead of outputting probabilities it outputs a class identity, which depends on the sign of $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$. SVMs make use of the **kernel trick**, the "trick" being that we can rewrite $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$ completely in terms of dot products between examples. The general form of our prediction function becomes

If our kernel function is just $k(x, x^{(i)}) = x^T x^{(i)}$ then we've just rewritten $\boldsymbol{w}$ in the form $\boldsymbol{w} \to \boldsymbol{X}^T\boldsymbol{\alpha}$

$$f(\boldsymbol{x}) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) \tag{5.83}$$

where the *kernel* [function] takes the general form $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$. A major drawback to kernel machines (methods) in general is that the cost of evaluating the decision function $f(\boldsymbol{x})$ is linear in the number of training examples. SVMs, however, are able to mitigate this by learning an α with mostly zeros. The training examples with *nonzero* α_i are known as **support vectors**.

Deep Networks: Modern Practices

Contents

2.1 Deep Feedforward Networks (Ch. 6) 24

2.1.1 Back-Propagation (6.5) 25

2.2 Regularization for Deep Learning (Ch. 7) 26

2.3 Optimization for Training Deep Models (Ch. 8) 28

2.4 Convolutional Neural Networks (Ch. 9) 32

2.5 Sequence Modeling (RNNs) (Ch. 10) 35

2.5.1 Review: The Basics of RNNs 35

2.5.2 RNNs as Directed Graphical Models 40

2.5.3 Challenge of Long-Term Deps. (10.7) 42

2.5.4 LSTMs and Other Gated RNNs (10.10) 43

2.6 Applications (Ch. 12) 44

2.6.1 Natural Language Processing (12.4) 44

2.6.2 Neural Language Models (12.4.2) 45

Deep Feedforward Networks (Ch. 6)

Table of Contents Local *Written by Brandon McKinzie*

The strategy/purpose of [feedforward] deep learning is to *learn the set of features/representation describing* $\boldsymbol{x}$ with a mapping ϕ before applying a linear model. In this approach, we have a model

$$y = f(\boldsymbol{x};\ \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$$

with ϕ defining a hidden layer.

ReLUs and their generalizations. Some nice properties of ReLUs are...

Recall the ReLU activation function: $g(z) = \max\{0, z\}$.
a.e. is short for "almost everywhere"

- Derivatives through a ReLU remain large and consistent whenever the unit is active.
- Second derivative is 0 a.e. and the derivative is 1 everywhere the unit is active, meaning the gradient direction is more useful for learning than it would be with activation functions that introduce 2nd-order effects (see equation 4.9)

Generalizing to aid gradients when $\boldsymbol{z} < 0$. Three such generalizations are based on using a nonzero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i) \tag{57}$$

→ Absolute value rectification: fix $\alpha_i = -1$ to obtain $g(z) = |z|$.
→ Leaky ReLU: fix α_i to a small value like 0.01.
→ Parametric ReLU (PReLU): treats α_i like a learnable parameter.

Logistic sigmoid and hyperbolic tangent. Sigmoid activations on hidden units is a bad idea, since they're only sensitive to their inputs near zero, with small gradients everywhere else. If sigmoid activations must be used, tanh is probably a better substitute, since it resembles the identity (i.e. a linear function) near zero.

Back-Propagation (6.5)

The chain rule. Suppose $\boldsymbol{z} = f(\boldsymbol{y})$ where $\boldsymbol{y} = g(\boldsymbol{x})$ (see margin for dimensions). Then[9],

$\boldsymbol{x} \in \mathbb{R}^m$

$\boldsymbol{y} \in \mathbb{R}^n$

$z : \mathbb{R}^n \to \mathbb{R}$

$g : \mathbb{R}^m \to \mathbb{R}^n$

$$\frac{\partial z}{\partial x_i} = (\nabla_{\boldsymbol{x}} z)_i = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{n} (\nabla_{\boldsymbol{y}} z)_j \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{n} (\nabla_{\boldsymbol{y}} z)_j \ (\nabla_{\boldsymbol{x}} y_j)_i \tag{6.45}$$

$$\to \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z = \boldsymbol{J}^T_{y=g(\boldsymbol{x})} \nabla_{\boldsymbol{y}} z \tag{6.46}$$

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$.

[9] Note that we can view $z = f(\boldsymbol{y})$ as a multi-variable function of the dimensions of $\boldsymbol{y}$,

$$z = f(y_1, y_2, \ldots, y_n)$$

Regularization for Deep Learning (Ch. 7)

 Written by Brandon McKinzie

Recall the definition of regularization: "any modification we make to a learning algorithm that is intented to reduce its generalization error but not its training error."

Limiting Model Capacity. Recall that **Capacity** [of a model] is the ability to fit a wide variety of functions. Low cap models may struggle to fit training set, while high cap models may overfit by simply memorizing the training set. We can limit model capacity by adding a parameter norm penalty $\Omega(\theta)$ to the objective function J:

$$\widetilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta) \quad \text{where} \quad \alpha \in [0, \infty) \tag{7.1}$$

where we typically choose Ω to only penalize the *weights* and leave biases unregularized.

L2-Regularization. Defined as setting $\Omega(\theta) = \frac{1}{2}||w||_2^2$. Assume that $J(w)$ is quadratic, with minimum at w^*. Since quadratic, we can approximate J with a second-order expansion about w^*.

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \tag{7.6}$$

$$\nabla_w \hat{J}(w) = H(w - w^*) \tag{7.7}$$

where $H_{ij} = \frac{\partial^2 J}{\partial w_i w_j}\big|_{w^*}$. If we add in the [derivative of] the weight decay and set to zero, we obtain the solution

$$\widetilde{w} = (H + \alpha I)^{-1} H w^* \tag{7.10}$$

$$= Q(\Lambda + \alpha I)^{-1} \Lambda Q^T w^* \tag{7.13}$$

which shows that the effect of regularization is to rescale the i eigenvectors of H by $\frac{\lambda_i}{\lambda_i + \alpha}$. This means that eigenvectors with $\lambda_i >> \alpha$ are relatively unchanged, but the eigenvectors with $\lambda_i << \alpha$ are shrunk to nearly zero. In other words, only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact.

Sparse Representations. Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the *activations* of the units, encouraging their activations to be sparse. It's important to distinguish the difference between sparse parameters and sparse *representations*. In the former, if we take the example of some $\boldsymbol{y} = \boldsymbol{B}\boldsymbol{h}$, there are many zero entries in some parameter matrix $\boldsymbol{B}$ while, in the latter, there are many zero entries in the representation vector $\boldsymbol{h}$. The modification to the loss function, analogous to 7.1, takes the form

$$\widetilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h}) \quad \text{where} \quad \alpha \in [0, \infty) \tag{7.48}$$

Adversarial Training Even for networks that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally constructed to search for an input $\boldsymbol{x}'$ near a data point $\boldsymbol{x}$ such that the model output for $\boldsymbol{x}'$ is very different than the output for $\boldsymbol{x}$.

In many cases, $\boldsymbol{x}'$ can be so similar to $\boldsymbol{x}$ that a human cannot tell the difference!

$$\boldsymbol{x}' \longleftarrow \boldsymbol{x} + \epsilon \cdot \text{sign}\left(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})\right) \tag{58}$$

In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed training examples.

Empirical Risk Minimization. The ultimate goal of any machine learning algorithm is to reduce the expected generalization error, also called the **risk**:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}}\left[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)\right] \tag{59}$$

risk definitions

with emphasis that the risk is over the *true* underlying data distribution p_{data}. If we knew p_{data}, this would be an optimization problem. Since we don't, and only have a set of training samples, it is a machine learning problem. However, we can still just minimize the **empirical risk**, replacing p_{data} in the equation above with $\hat{p}_{data}$[10].

So, how is minimizing the empirical risk any different than familiar gradient descent approaches? Aren't they designed to do just that? Well, sort of, but it's technically not the same. When we say "minimize the empirical risk" in the context of optimization, we mean this very literally. Gradient descent methods emphatically do *not* just go and *set* the weights to values such that the empirical risk reaches its lowest possible value – that's not machine learning. Furthermore, many useful loss function such as 0-1 loss[11] do not have useful derivatives.

ERM $\neq$ GD

Surrogate Loss Functions and Early Stopping. In cases such as 0-1 loss, where minimization is intractable, one typically optimizes a **surrogate loss function** instead, such as the negative log-likelihood of the correct class. Also, an important difference between pure optimization and our training algorithms is that the latter usually don't halt at a local minimum. Instead, we usually must define some early stopping condition to terminate training before overfitting begins to occur.

> *We want to minimize the risk, but we don't have access to p_{data}, so ...*
> *We want to minimize the empirical risk, but it's prone to overfitting and our loss function's derivative may be zero/undefined, so ...*
> *We minimize a surrogate loss function iteratively over minibatches until early stopping is triggered.*

[10] This amount to a simple average over the loss function at each training point.

[11] The 0-1 loss function is defined as

$$L(\hat{y}, y) = I(\hat{y} \neq y) \tag{60}$$

Batch and Minibatch Algorithms. Computing $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ as an expectation over the entire training set is expensive, so we typically compute the expectation over a small subset of the examples. Recall that the standard deviation, or standard error $SE(\mu_m)$, of the mean taken over some subset of $m \leq n$ samples, $\mu_m = \frac{1}{m} \sum_{i \sim Rand(0,\ n,\ size=m)} x^{(i)}$, is given by $\sigma/\sqrt{m}$, where σ is the true [sample] standard deviation of the full n data samples. In other words, to improve such a gradient by a factor of 10 requires 100 times more samples-per-batch (and thus 100 times more computation). For this reason, most optimization algorithms actually *converge* much faster if they can rapidly compute approximate estimates of the gradient (re: smaller batches) rather than slowly computing the exact gradient.

The key points to consider when choosing your batch size:

1. Larger batches = more accurate estimates of the gradient, but with less than linear returns.
2. If examples in the batch are processed in parallel (as is typical), then memory roughly scales with batch size.
3. Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1. However, this requires a low learning rate to maintain stability and thus a longer overall training runtime.

Also, note that online SGD, where we never reuse data points, but simply update parameters as new data comes in, gives an unbiased estimator of the exact gradient of the generalization error (the risk). Once data samples are reused (e.g. when training with multiple epochs), the gradient estimates become biased. The interesting point here is that the availability of increasingly massive datasets is making single-epoch[12] training more common. In such cases, *overfitting is no longer an issue*, but rather underfitting and computational efficiency.

Ill-conditioning of the Hessian matrix $\boldsymbol{H}$ can cause SGD to get "stuck" in the sense that even very small steps increase the cost function. Recall that a second-order Taylor series expansion of the cost function predicts that an SGD step of $-\epsilon \boldsymbol{g}$ will add

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} - \epsilon \boldsymbol{g}^T \boldsymbol{g} \tag{61}$$

to the cost. If $\boldsymbol{H}$ has a large condition number (re: if $\boldsymbol{H}$ is ill-conditioned), then the range of possible values, $[1/\lambda_{max}, 1/\lambda_{min}]$, for $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ can become very large. In particular, if $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ exceeds $\epsilon \boldsymbol{g}^T \boldsymbol{g}$, then the SGD step will *increase* the cost!

[12] Or even less, i.e. not using all of the training data.

Training algorithms. Below, I list some popular training algorithms and their update equations.

- **SGD**.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{62}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{63}$$

- **Momentum**.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{64}$$

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g} \tag{65}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{66}$$

- **Nesterov Momentum**. Gradient computations instead evaluated after current velocity is applied.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)}\right) \tag{67}$$

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g} \tag{68}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{69}$$

- **AdaGrad**. Different learning rate for each model parameter. Individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. Empirically, can result in premature and excessive decrease in the effective learning rate.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{70}$$

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g} \tag{71}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g} \tag{72}$$

  where the gradient accumulation variable $\boldsymbol{r}$ is initialized to the zero vector, and the fraction and square root in the last equation is applied element-wise.

- **RMSProp**. Modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{73}$$

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g} \tag{74}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g} \tag{75}$$

  It is also common to modify RMSProp to use Nesterov momentum.

- **Adam**. So-named to mean "adaptive moments." We now call $\boldsymbol{r}$ the 2nd moment (variance) variable, and introduce $\boldsymbol{s}$ as the 1st moment (mean) variable, where the moments are for the [true] gradient; the new variables act as estimates of the moments [since we estimate the gradient with a simple average over a minibatch]. Note that these moments are uncentered.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{76}$$

$$\boldsymbol{s} \leftarrow \frac{1}{1-\rho_1^{t-1}} \left[\rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}\right] \tag{77}$$

$$\boldsymbol{r} \leftarrow \frac{1}{1-\rho_2^{t-1}} \left[\rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}\right] \tag{78}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{s} \tag{79}$$

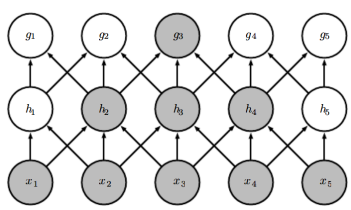
where the factors proportional to the ρ values serve to correct for bias in the moment estimators.

We use a 2-D image I as our input (and therefore require a 2-D kernel K). Note that most neural networks do not technically implement convolution[13], but instead implement a related function called the *cross-correlation*, defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{9.6}$$

Convolution leverages the following three important ideas:

- **Sparse interactions[/connectivity/weights]**. Individual input units only interact/-connect with a subset of the output units. Accomplished by making the kernel smaller than the input. It's important to recognize that the receptive field of the units in the deeper layers of a convolutional network is *larger* than the receptive field of the units in the shallow layers, as seen below.



- **Parameter sharing**.
- **Equivariance** (to translation). Changes in inputs [to a function] cause output to change in the same way. Specifically, f is equivariant to g if $f(g(x)) = g(f(x))$. For convolution, g would be some function that translates the input.

[13] Technically the convolution output is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{9.4}$$

$$= (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{9.5}$$

where 9.5 can be asserted due to commutativity of convolution.

Pooling. Helps make the representation approximately **invariant** to small translations of the input. The use of pooling can be viewed as adding an infinitely strong prior[14] that the function the layer learns must be invariant to small translations.

Additional common tricks[15].

- **Local Response Normalization (LRN)**[16]. Purpose is to aid generalization ability. Let $a^i_{x,y}$ denote the activity of a neuron computed by applying kernel i at position (x, y) and then applying the ReLU nonlinearity. The response-normalized activity $b^i_{x,y}$ is given by the expression

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum_j \left(a^j_{x,y}\right)^2\right)^\beta} \tag{80}$$

  where j runs from $[i - n/2]_+$ to $\min(N-1,\ i+n/2)$, and N is the total number of kernels in the given layer[17]. Authors used $k = 2$, $n = 5$, $\alpha = 10^{-4}$, $\beta = 0.75$.
- **Batch Normalization**[18]. BN Allows us to use much higher learning rates and be less careful about initialization. Algorithm defined in image below, where each element of the batch, $x_i \equiv x_i^{(k)}$ (where we drop the k for notational simplicity), represents the kth activation output from the previous layer [for the ith sample in the batch] and about to be fed as input to the current layer.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^m x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^m (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

  Note that one can model each layer's activations as arising from some distribution. When we feed data to a network, we model the data as coming from some data-generating

[14] Where the distribution of this prior is over all possible functions learnable by the model.

[15] Collected on my own. In other words, not from the deep learning book, but rather a bunch of disconnected resources over time.

[16] From section 3.3 of Krizhevsky et al. (2012). AlexNet paper.

[17] In other words, the summation is over the adjacent kernel maps, with [total] window size n (manually chosen). The min/max just says $n/2$ to the left (right) unless that would be past the leftmost (rightmost) kernel map.

[18] From "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" by Ioffe et al.

distribution. Similarly, we can model the activations that occur when feeding the data through as coming from some activation-layer-generating distribution. The problem with this model is that the process of updating our weights during training changes the distribution of activations for each layer, which can make the learning task more difficult. Batch normalization's goal is to reduce this *internal covariate shift*, and is motivated by the practice of normalizing our data to have zero mean and unit variance.

Intuition of some math.

- **Q**: How to intuitively understand the commutativity of convolution?
  - **A**: You must first realize that, *independent of which formula we're thinking of*, as one index into either the image or kernel increases (decreases), the other decreases (increases); they increment in opposite directions. The difference between the two formulas (9.4 and 9.5 in textbook), is just that we start at different ends of the summations (when you actually substitute in the numbers). Note that this doesn't require any symmetry on the boundaries of the summation about zero; it truly is a property of the convolution.
- **Q**: What do the authors mean by "we have flipped the kernel"?
  - **A**: Not much, and it's poor wording. They didn't *do* anything, that is just part of the definition of the convolution. They literally just mean that the convolution has the property that as one index increases, the other decreases (see previous answer). The cross-correlation, however, has the property that as one index increases, the other increases, too.

Modern Practices **January 15**

Sequence Modeling (RNNs) (Ch. 10)

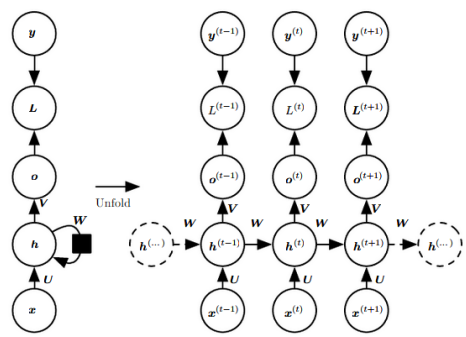
Table of Contents Local *Written by Brandon McKinzie*

Review: The Basics of RNNs

Notation/Architecture Used.

- **U**: input → hidden.
- **W**: hidden → hidden.
- **V**: hidden → output.
- **Activations**: tanh [hidden] and softmax [after output].
- **Misc. Details**: $\boldsymbol{x}^{(t)}$ is a *vector* of inputs fed at time t. Recall that RNNs can be unfolded for any desired number of steps τ. For example, if $\tau = 3$, the general functional representation output of an RNN is $\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)};\ \boldsymbol{\theta}) = f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta});\ \boldsymbol{\theta})$. Typical RNNs read information out of the state $\boldsymbol{h}$ to make predictions.

Shape of $\boldsymbol{x}^{(t)}$ fixed, e.g. vocab length.



Black square on recurrent connection ≡ interaction w/delay of a single time step.

Forward Propagation & Loss. Specify initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, feed input sequence $\boldsymbol{x}^{(t)}$ and compute the output sequence $\boldsymbol{o}^{(t)}$. To determine the loss at each time-step, $L^{(t)}$, we compare $\text{softmax}(\boldsymbol{o}^{(t)})$ with (one-hot) $\boldsymbol{y}^{(t)}$.

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}) \qquad \text{where} \qquad \boldsymbol{a}^{(t)} = b + W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} \tag{10.9/8}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}) \qquad \text{where} \qquad \boldsymbol{o}^{(t)} = c + V\boldsymbol{h}^{(t)} \tag{10.11/10}$$

Note that this is an example of an RNN that maps input seqs to output seqs of the same length[19]. We can then compute, e.g., the log-likelihood loss $L = \sum_t L^{(t)}$ over all time steps as:

$$L = -\sum_t \log\left(p_{model}\left[y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right]\right) \tag{10.12/13/14}$$

Convince yourself this is identical to cross-entropy.

[19] Where "same length" is related to the number of timesteps (i.e. τ input steps means τ output steps), not anything about the actual shapes/sizes of each individual input/output.

where $y^{(t)}$ is the **ground-truth** (one-hot vector) at time t, whose probability of occurring is given by the corresponding element of $\hat{\boldsymbol{y}}^{(t)}$

Back-Propagation Through Time.

1. **Internal-Node Gradients**. In what follows, when considering what is included in the chain rule(s) for gradients with respect to a node $\boldsymbol{N}$, just need to consider paths from it [through its **descendents**] to loss node(s).
   - **Output nodes**. For any given time t, the node $\boldsymbol{o}^{(t)}$ has only one direct descendant, the loss node $L^{(t)}$. Since no other loss nodes can be reached from $\boldsymbol{o}^{(t)}$, it is the only one we need consider in the gradient.

$$
\begin{aligned}
\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i &= \frac{\partial L}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial L}{\partial L^{(t)}} \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= (1) \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ -\log\left(\hat{\boldsymbol{y}}_{y^{(t)}}^{(t)}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \log\left(\frac{e^{\boldsymbol{o}_{y^{(t)}}^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \boldsymbol{o}_{y^{(t)}}^{(t)} - \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\}
\end{aligned}
\tag{81}
$$

Ground-truth $y^{(t)}$ here is a **scalar**, interpreted as the index of the correct label of output vector.

$$\mathbf{1}_{i,y^{(t)}} = \begin{cases} 1 & y^{(t)} = i \\ 0 & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{1}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \frac{\partial \sum_j e^{\boldsymbol{o}_j^{(t)}}}{\partial \boldsymbol{o}_i^{(t)}} \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{e^{\boldsymbol{o}_i^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \hat{\boldsymbol{y}}_i^{(t)} \right\} \\
&= \hat{\boldsymbol{y}}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}
\end{aligned}
\tag{10.18}
$$

which leaves all entries of $\boldsymbol{o}^{(t)}$ unchanged *except* for the entry corresponding to the true label, which will become negative in the gradient. All this means is, since we

want to increase the probability of this entry, driving this value up will *decrease* the loss (hence negative) and driving any other entries up will *increase* the loss proportional to its current estimated probability (driving up an [incorrect] entry that is already high is "worse" than driving up a small [incorrect entry]).

- **Hidden nodes**. First, consider the simplest hidden node to take the gradient of, the last one, $\boldsymbol{h}^{(\tau)}$ (simplest because only one descendant [path] reaching any loss node(s)).

$$
\begin{aligned}
\left(\nabla_{\boldsymbol{h}^{(\tau)}} L\right)_i &= \frac{\partial L}{\partial L^{(\tau)}} \sum_{k=1}^{n_{out}} \frac{\partial L^{(\tau)}}{\partial \boldsymbol{o}_k^{(\tau)}} \frac{\partial \boldsymbol{o}_k^{(\tau)}}{\partial \boldsymbol{h}_i^{(\tau)}} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \frac{\partial \boldsymbol{o}_k^{(\tau)}}{\partial \boldsymbol{h}_i^{(\tau)}} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \frac{\partial}{\partial \boldsymbol{h}_i^{(\tau)}} \left\{ c_k + \sum_{j=1}^{n_{hid}} V_{kj} \boldsymbol{h}_j^{(\tau)} \right\} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k V_{ki} \\
&= \sum_{k=1}^{n_{out}} (V^T)_{ik} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \\
&= \left(V^T \nabla_{\boldsymbol{o}^{(t)}} L\right)_i
\end{aligned}
\tag{10.19}
$$

Before proceeding, **notice the following useful pattern**: If two nodes a and b, each containing n_a and n_b neurons, are fully connected by parameter matrix $W_{n_b \times n_a}$ and directed like $a \to b \to L$, then[20] $\nabla_a L = W^T \nabla_b L$. Using this result, we can then iterate and take gradients back in time from $t = \tau - 1$ to $t = 1$ as follows:

$$
\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \tag{10.20}
$$

$$
\begin{aligned}
&= W^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) \operatorname{diag}(1 - \tanh^2(\boldsymbol{a}^{(t+1)})) + V^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \\
&= W^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t+1)})^2\right) + V^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)
\end{aligned}
\tag{10.21}
$$

$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$

$(\operatorname{diag}(\boldsymbol{a}))_{ii} \triangleq a_i$

2. **Parameter Gradients**. Now we can compute the gradients for the parameter matrices/vectors, where it is crucial to remember that a given parameter matrix (e.g. U) is shared across *all* time steps t. We can treat tensor derivatives in the same form as

[20] More generally,

$$\nabla_a L = (\frac{\partial b}{\partial a})^T \nabla_b L$$

which is a good example of how vector derivatives map into a matrix. For example, let $\boldsymbol{a} \in \mathbb{R}^{n_a}$ and $\boldsymbol{b} \in \mathbb{R}^{n_b}$. Then

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \in \mathbb{R}^{n_b \times n_a}$$

previously done with vectors after a quick abstraction: For any tensor $\mathbf{X}$ of arbitrary rank (e.g. if rank-4 then index like $\mathbf{X}_{ijk\ell}$), use single variable (e.g. i) to represent the complete tuple of indices[21].

- **Bias parameters [vectors]**. These are nothing new, since just vectors.

$$\begin{aligned}(\nabla_{\boldsymbol{c}} L) &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \\ &= \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)\end{aligned} \tag{10.22}$$

$$\begin{aligned}(\nabla_{\boldsymbol{c}} L) &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \\ &= \sum_t \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)\end{aligned} \tag{10.23}$$

- **$\mathbf{V}$ ($n_{out} \times n_{hid}$)**.

$$\nabla_{\mathbf{V}} L = \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)} \tag{82a}$$

$$= \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)}(\boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_{n_{out}}^{(t)}) \tag{82b}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{82c}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \left\{ c_i + \sum_{j=1}^{n_{hid}} V_{ij} \boldsymbol{h}_j^{(t)} \right\} \tag{82d}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \begin{bmatrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{h}_1^{(t)} & \boldsymbol{h}_2^{(t)} & \ldots & \boldsymbol{h}_{n_{hid}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{82e}$$

$$= \sum_t^{\tau} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) (\boldsymbol{h}^{(t)})^T \tag{82f}$$

[21] More details on tensor derivatives: Consider the chain defined by $\mathbf{Y} = g(\mathbf{X})$, and $z = f(\mathbf{Y})$, where z is some vector. Then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

where if 82e confuses you, see the footnote[22].

- **W** $(n_{hid} \times n_{hid})$. This one is a bit odd, since $\boldsymbol{W}$ is, in a sense, even more "shared" across time steps than $\boldsymbol{V}$[23]. The authors here define/choose, when evaluating $\nabla_{\mathbf{W}} h_i^{(t)}$ to only concern themselves with $\boldsymbol{W} := \boldsymbol{W}^{(t)}$, i.e. the direct connections to $\boldsymbol{h}$ at time t.

$$\nabla_{\mathbf{W}} L = \sum_t^{\tau} \nabla_{\mathbf{W}} L^{(t)} \tag{84a}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \nabla_{\mathbf{W}^{(\mathbf{t})}} \boldsymbol{h}_i^{(t)} \tag{10.25}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \left(\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \begin{bmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ \boldsymbol{h}_1^{(t-1)} & \boldsymbol{h}_2^{(t-1)} & \dots & \boldsymbol{h}_{n_{hid}}^{(t-1)} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \tag{84b}$$

$$= \sum_t^{\tau} \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) (\boldsymbol{h}^{(t-1)})^T \tag{10.26}$$

- **U** $(n_{hid} \times n_{in})$. Very similar to the previous calculation.

$$\nabla_{\mathbf{U}} L = \sum_t^{\tau} \nabla_{\mathbf{U}} L^{(t)} \tag{85a}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \nabla_{\mathbf{U}^{(\mathbf{t})}} \boldsymbol{h}_i^{(t)} \tag{10.27}$$

$$= \sum_t^{\tau} \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) (\boldsymbol{x}^{(t)})^T \tag{10.28}$$

---

[22] The general lesson learned here is that, for some matrix $\boldsymbol{W} \in \mathbb{R}^{a \times b}$ and vector $\boldsymbol{x} \in \mathbb{R}^b$,

$$\sum_i \nabla_{\mathbf{W}} \left[(\boldsymbol{W}x)_i\right] = \begin{bmatrix} \boldsymbol{x}^T \\ \boldsymbol{x}^T \\ \vdots \\ \boldsymbol{x}^T \end{bmatrix} \tag{83}$$

where, of course, the output has the same dimensions as $\boldsymbol{W}$.

[23] Specifically, $\boldsymbol{h}^{(t)}$ is both

(a) An explicit function of the parameter matrix $\boldsymbol{W}^{(t)}$ directly feeding into it.

(b) An implicit function of all other $\boldsymbol{W}^{t=i}$ that came before.

This is different than before, where we had $\boldsymbol{o}^{(t)}$ not implicitly depending on earlier $\boldsymbol{V}^{(t=i)}$. In other words, $\boldsymbol{h}^{(t)}$ is a <u>descendant</u> of all earlier (and current) $\boldsymbol{W}$.

RNNs as Directed Graphical Models

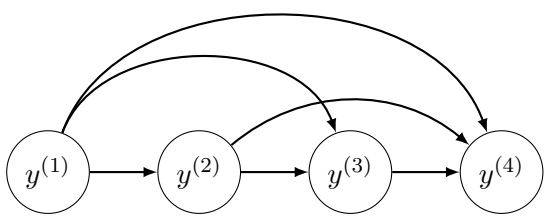
The advantage of RNNs is their efficient parameterization of the joint distribution over $\boldsymbol{y}^{(i)}$ via parameter sharing. This introduces a built-in assumption that we can model the effect of $y^{(i)}$ in the distant past on the current $y^{(t)}$ *via its effect on* $\boldsymbol{h}$. We are also assuming that the conditional probability distribution over the variables at $t+1$ given the variables at time t is **stationary**. Next, we want to know how to draw *samples* from such a model. Specifically, how to sample from the conditional distribution ($y^{(t)}$ given $y^{(t-1)}$) at each time step.

Say we want to model a sequence of scalar random variables $\mathbb{Y} \triangleq \{y^{(1)}, \ldots, y^{(\tau)}\}$ for some sequence length τ. Without making independence assumptions just yet, we can parameterize the joint distribution $P(\mathbb{Y})$ with basic definitions of probability:

$$P(\mathbb{Y}) \triangleq P(y^{(1)}, \ldots, y^{(\tau)}) = \prod_{t=1}^{\tau} P(y^{(t)} \mid y^{(t-1)}, \ldots, y^{(1)}) \tag{86}$$

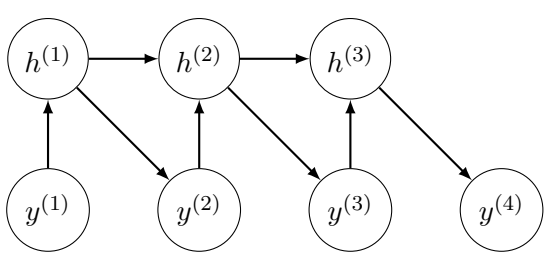
where I've drawn an example of the *complete graph* for $\tau = 4$ below.

The complete graph can represent the direct dependencies between any pairs of y values.



If each value y could take on the same fixed set of k values, we would need to learn k^4 parameters to represent the joint distribution $P(\mathbb{Y})$. This clearly inefficient, since the number of parameters needed scales like $\mathcal{O}(k^\tau)$. If we relax the restriction that each $y^{(i)}$ must depend *directly* on all past $y^{(j)}$, we can considerably reduce the number of parameters needed to compute the probability of some particular sequence.

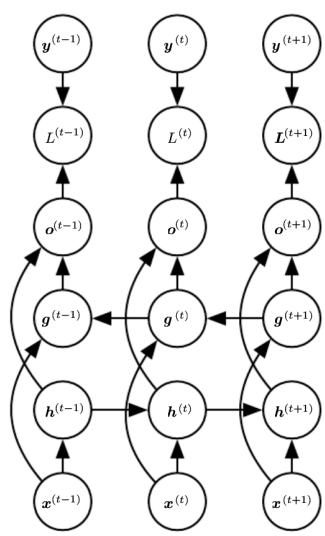
We could include latent variables h at each timestep that capture the dependencies, reminiscent of a classic RNN:



Since in the RNN case all factors $P(h^{(t)} \mid h^{(t-1)})$ are deterministic, we don't need any additional parameters to compute this probability[24], other than the single m^2 parameters needed to convert any $h^{(t)}$ to the next $h^{(t+1)}$ (which is shared across all transitions). Now, the number of parameters needed as a function of sequence length is constant, and as a function of k is just $\mathcal{O}(k)$.

Finally, to view the RNN as a graphical model, we must describe how to sample from it, namely how to sample a sequence $\boldsymbol{y}$ from $P(\mathbb{Y})$, if parameterized by our graphical model above. In the general case where we don't know the value of τ for our sequence $\boldsymbol{y}$, one approach is to have a EOS symbol that, if found during sampling, means we should stop there. Also, in the typical case where we actually want to model $P(y \mid x)$ for input sequence x, we can reinterpret the parameters $\boldsymbol{\theta}$ of our graphical model as a function of $\boldsymbol{x}$ the input sequence. In other words, the graphical model interpretation becomes a function of $\boldsymbol{x}$, where $\boldsymbol{x}$ determines the exact values of the probabilities the graphical model takes on – an "instance" of the graphical model.

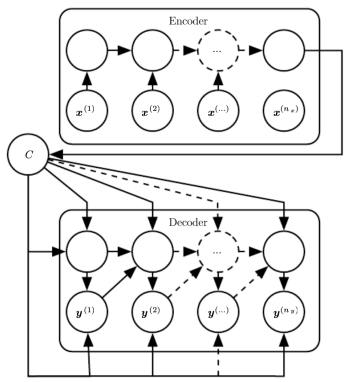
Bidirectional RNNs. In many applications, it is desirable to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on *the whole sequence*. For example, in speech recognition, the interpretation of words/sentences can also depend on what is *about* to be said. Below is a typical bidirectional RNN, where the inputs $\boldsymbol{x}^(t)$ are fed both to a "forward" RNN ($\boldsymbol{h}$) and a "backward" RNN ($\boldsymbol{g}$).



Notice how the output units $\boldsymbol{o}^{(t)}$ have the nice property of depending on both the past and future while being most sensitive to input values around time t.

[24]Don't forget that, in a neural net, a variable $y^{(t)}$ is represented by a *layer*, which itself is composed of k nodes, each associated with one of the k unique values that $y^{(t)}$ could be.

Encoder-Decoder Seq2Seq Architectures (10.4) Here we discuss how an RNN can be trained to map an input sequence to output sequence which is not necessarily the same length. (Not really much of a discussion...figure below says everything.)



Challenge of Long-Term Deps. (10.7)

Gradients propagated over many stages either vanish (usually) or explode. We saw how this could occur when we took parameter gradients earlier, and for weight matrices $\boldsymbol{W}$ further along from the loss node, the expression for $\nabla_{\mathbf{W}} L$ contained multiplicative Jacobian factors. Consider the (linear activation) repeated function composition of an RNN's hidden state in 10.36. We can rewrite it as a power method (10.37), and if $\boldsymbol{W}$ admits an eigendecomposition (remember $\boldsymbol{W}$ is necessarily square here), we can further simplify as seen in 10.38.

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^T \boldsymbol{h}^{(t-1)} \tag{10.36}$$
$$= (\boldsymbol{W}^t)^T \boldsymbol{h}^{(0)} \tag{10.37}$$
$$= \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)} \tag{10.38}$$

Q: Explain interp. of mult. $\boldsymbol{h}$ by $\boldsymbol{Q}$ as opposed to the usual $\boldsymbol{Q}^T$ explained in the linear algebra review.

> **Any component of $\boldsymbol{h}^{(0)}$ that isn't aligned with the largest eigenvector will eventually be discarded.**[25]

If, however, we have a non-recurrent network such that the state elements are repeatedly multiplied by different $w^{(t)}$ at each time step, the situation is different. Suppose the different $w^{(t)}$ are i.i.d. with mean 0 and variance v. The variance of the product is easily seen to

[25] Make sure to think about this from the right perspective. The largest value of $t = \tau$ in the RNNs we've seen would correspond with either (1) the largest output sequence or (2) the largest input sequence (if fixed-vector output). After we extract the output from a given forward pass, we reset the clock and either back-propagate errors (if training) or get ready to feed another sequence.

be $\mathcal{O}(v^n)$[26]. *To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$.*

LSTMs and Other Gated RNNs (10.10)

While leaky units have connection weights that are either manually chosen constants or are trainable parameters, gated RNNs generalize this to connection weights that may change *at each time step*. Furthermore, gated RNNs can learn to both accumulate and *forget*, while leaky units are designed for just accumulation[27]

LSTM (10.10.1). The idea is we want self-loops to produce paths where the gradient can flow for long durations. The self-loop weights are **gated**, meaning they are controlled by another hidden unit, interpreted as being conditioned on *context*. Listed below are the main components of the LSTM architecture.

The subscript, i, identifies the cell. The superscript, t, denotes the time.

- **Forget gate** $f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$.
- **Internal state** $s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + g_i^{(t)} \odot \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$.
- **External input gate** $g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$.
- **Output gate** $q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$.

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{89}$$

[26]Quick sketch of (my) proof:

$$\mathrm{Var}\left[w^{(i)}\right] = v = \mathbb{E}\left[(w^{(i)})^2\right] - \cancel{\mathbb{E}\left[w^{(i)}\right]^2} \tag{87}$$

$$\mathrm{Var}\left[\prod_t^n w^{(t)}\right] = \mathbb{E}\left[\left(\prod_t^n w^{(t)}\right)^2\right] = \prod_t^n \mathbb{E}\left[(w^{(t)})^2\right] = v^n \tag{88}$$

[27]Q: Isn't choosing to update with higher relative weight on the present the same as forgetting? A: Sort of. It's like "soft forgetting" and will inevitably erase more/less than desired (smeary). In this context, "forget" means to set the weight of a specific past cell to zero.

Natural Language Processing (12.4)

Begins on pg. 448

n-grams. A **language model** defines a probability distribution over sequences of [discrete] tokens (words/characters/etc). Early models were based on the *n-gram*: a [fixed-length] sequence of n tokens. Such models define the conditional distribution for the nth token, given the $(n-1)$ previous tokens:

$$P(x_t \mid x_{t-(n-1)}, \ldots, x_{t-1})$$

where x_i denotes the token at step/index/position i in the sequence.

To define distributions over longer sequences, we can just use Bayes rule over the shorter distributions, as usual. For example, say we want to find the [joint] distribution for some τ-gram ($\tau > n$), and we have access to an n-gram model and a [perhaps different] model for the initial sequence $P(x_1, \ldots, x_{n-1})$. We compute the τ distribution simply as follows:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-1}, \ldots, x_{t-(n-2)}, x_{t-(n-1)}) \tag{12.5}$$

where it's important to see that each factor in the product is a distribution over a length-n sequence. Since we need that initial factor, it is common to train both an n-gram model and an $n-1$-gram model simultaneously.

Let's do a specific example for a trigram (n = 3).

- **Assumptions [for this trigram model example]**:
  - For any $n \geq 3$, $P(x_n \mid x_1, \ldots, x_{n-1}) = P(x_n \mid x_{n-2}, x_{n-1})$.
  - When we get to computing the full joint distribution over some sequence of arbitrary length, we assume we have access to both P_3 and P_2, the joint distributions over all subsequences of length 3 and 2, respectively.
- **Example sequence**: We want to know how to use a trigram model on the sequence ['THE', 'DOG', 'RAN', 'AWAY'].

- **Derivation**: We can use the built-in model assumption to derive the following formula.

$$\begin{aligned} P(\text{THE DOG RAN AWAY}) &= P_3(\text{AWAY} \mid \text{THE DOG RAN})\ P_3(\text{THE DOG RAN}) \\ &= P_3(\text{AWAY} \mid \text{DOG RAN})\ P_3(\text{THE DOG RAN}) \\ &= \frac{P_3(\text{DOG RAN AWAY})}{P_2(\text{DOG RAN})}\ P_3(\text{THE DOG RAN}) \\ &= P_3(\text{THE DOG RAN}) P_3(\text{DOG RAN AWAY}) / P_2(\text{DOG RAN}) \end{aligned} \tag{12.7}$$

Limitations of n-gram. The last example illustrates some potential problems one may encounter that arise [if using MLE] when the full joint we seek is nonzero, but (a) some P_n factor is zero, or (b) P_{n-1} is zero. Some methods of dealing with this are as follows.

Recall that, in MLE, the P_n and P_{n-1} are usually approximated via counting occurrences in the training set

- **Smoothing**: shifting probability mass from the observed tuples to unobserved ones that are similar.
- **Back-off methods**: look up the lower-order (lower values of n) n-grams if the frequency of the context $x_{t-1}, \ldots, x_{t-(n-1)}$ is too small to use the higher-order model.

In addition, n-gram models are vulnerable to the curse of dimensionality, since most n-grams won't occur in the training set[28], even for modest n.

Neural Language Models (12.4.2)

Designed to overcome curse of dimensionality by using a distributed representation of words. Recognize that any model trained on sentences of length n and then told to generalize to new sentences [also of length n] must deal with a space[29] of possible sentences that is exponential in n. Such word representations (i.e. viewing words as existing in some high-dimensional space) are often called **word embeddings**. The idea is to map the words (or sentences) from the raw high-dimensional [vocab sized] space to a smaller feature space, where similar words are closer to one another. Using distributed representations may also be used with graphical models (think Bayes' nets) in the form multiple *latent variables*.

[28] For a given vocabulary, which usually has much more than n possible words, consider how many possible sequences of length n.

[29] Ok I tried re-wording that from the book's confusing wording but that was also a bit confusing. Let me break it down. Say you train on a thousand sentences each of length 5. For a given vocabulary of size VOCAB_SIZE, the number of possible sequences of length 5 is (VOCAB_SIZE)^5, which can be quite a lot more than a thousand (not to mention the possibility of duplicate training examples). To the naive model, all points in this high-dimensional space are basically the same. A neural language model, however, tries to arrange the space of possibilities in a meaningful way, so that an unforeseen sample at test time can be said "similar" as some previously seen training example. It does this by *embedding* words/sentences in a lower-dimensional feature space.

Deep Learning Research

Contents

3.1 Linear Factor Models (Ch. 13) 47

3.2 Autoencoders (Ch. 14) 50

3.3 Representation Learning (Ch. 15) 51

3.4 Structured Probabilistic Models for DL (Ch. 16) 52

3.4.1 Sampling from Graphical Models 54

3.4.2 Inference and Approximate Inference 54

3.5 Monte Carlo Methods (Ch. 17) 56

3.6 Confronting the Partition Function (Ch. 18) 58

3.7 Approximate Inference (Ch. 19) 59

3.8 Deep Generative Models (Ch. 20) 61

Linear Factor Models (Ch. 13)

 Written by Brandon McKinzie

Overview. Much research is in building a *probabilistic model*[30] of the input, $p_{model}(x)$. Why? Because then we can perform *inference* to predict stuff about our environment given any of the other variables. We call the other variables **latent variables**, h, with

$$p_{model}(x) = \sum_h \Pr(h) \Pr(x \mid h) = \mathbb{E}_h \left[p_{model}(x \mid h) \right] \tag{90}$$

So what? Well, the latent variables provide another means of *data representation*, which can be useful. **Linear factor models** (LFM) are some of the simplest probabilistic models with latent variables.

> A linear factor model is defined by the use of a stochastic linear decoder function that generates $\boldsymbol{x}$ by adding noise to a linear transformation of $\boldsymbol{h}$.

Note that $\boldsymbol{h}$ is a *vector* of arbitrary size, where we assume $p(\boldsymbol{h})$ is a **factorial distribution**: $p(\boldsymbol{h}) = \prod_i p(h_i)$. This roughly means we assume the elements of $\boldsymbol{h}$ are mutually independent[31]. The LFM describes the data-generation process as follows:

1. Sample the explanatory factors: $\boldsymbol{h} \sim p(\boldsymbol{h})$.
2. Sample the real-valued observable variables given the factors:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{91}$$

Probabilistic PCA and Factor Analysis.

- **Factor analysis**:

$$\boldsymbol{h} \sim \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I}) \tag{92}$$

$$\text{noise} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\psi} \equiv \text{diag}(\boldsymbol{\sigma^2})) \tag{93}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x};\ \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}) \tag{94}$$

  where the last relation can be shown by recalling that a linear combination of Gaussian variables is itself Gaussian, and showing that $\mathbb{E}_h[\boldsymbol{x}] = \boldsymbol{b}$, and $\text{Cov}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}$.

[30] Whereas, before, we've been building *functions* of the input (deterministic).

[31] Note that, technically, this assumption isn't strictly the definition of mutual independence, which requires that every *subset* (i.e. not just the full set) of $\{h_i \in \boldsymbol{h}\}$ follow this factorial property.

It is worth emphasizing the interpretation of $\boldsymbol{\psi}$ as the matrix of **conditional variances** σ_i^2. *Huh?* Let's take a step back. The fact that we were able to separate the distributions in the above relations for $\boldsymbol{h}$ and noise is from a built-in assumption that $\Pr(x_i|\boldsymbol{h}, x_{j\neq i}) = \Pr(x_i|\boldsymbol{h})$[32].

The Big Idea

The latent variable $\boldsymbol{h}$ is a big deal because it **captures the dependencies** between the elements of $\boldsymbol{x}$. *How do I know?* Because of our assumption that the x_i are conditionally independent given $\boldsymbol{h}$. If, once we specify $\boldsymbol{h}$, all the elements of $\boldsymbol{x}$ become independent, then any information about their interrelationship is hiding somewhere in $\boldsymbol{h}$.

Detailed walkthrough of Factor Analysis (a.k.a me slowly reviewing, months after taking this note):

- **Goal**. Analyze and understand the motivations behind how Factor Analysis defines the data-generation process under the framework of LFMs (defined in steps 1 and 2 earlier). Assume $\boldsymbol{h}$ has dimension n.
- **Prior**. Defines $p(\boldsymbol{h}) := \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I})$, the unit-variance Gaussian. Explicitly,

$$p(\boldsymbol{h}) := \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2}$$

- **Noise**. Assumed to be drawn from a Gaussian with diagonal covariance matrix $\boldsymbol{\psi} := \operatorname{diag}(\boldsymbol{\sigma}^2)$. Explicitly,

$$p(\text{noise} = \boldsymbol{a}) := \frac{1}{(2\pi)^{n/2}\prod_i^n \sigma_i} e^{-\frac{1}{2}\sum_i^n a_i^2/\sigma_i^2}$$

- **Deriving distribution of $\boldsymbol{x}$**. We use the fact that any linear combination of Gaussians is itself Gaussian. Thus, deriving $p(\boldsymbol{x})$ is reduced to computing it's mean and covariance matrix.

$$\begin{aligned}
\boldsymbol{\mu}_x &= \mathbb{E}_{\boldsymbol{h}}\left[\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}\right] && (95)\\
&= \int p(\boldsymbol{h})(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\mathrm{d}\boldsymbol{h} && (96)\\
&= \boldsymbol{b} + \int \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2} \boldsymbol{W}\boldsymbol{h}\mathrm{d}\boldsymbol{h} && (97)\\
&= \boldsymbol{b} && (98)\\
\operatorname{Cov}(\boldsymbol{x}) &= \mathbb{E}\left[(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])^T\right] && (99)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h} + \text{noise})(\boldsymbol{h}^T\boldsymbol{W}^T + \text{noise}^T)\right] && (100)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h}\boldsymbol{h}^T\boldsymbol{W}^T)\right] + \boldsymbol{\psi} && (101)\\
&= \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi} && (102)
\end{aligned}$$

where we compute the expectation of $\boldsymbol{x}$ over $\boldsymbol{h}$ because $\boldsymbol{x}$ is defined as a function of $\boldsymbol{h}$, and noise is always expectation zero.

[32] Due to <MATH>, this introduces a constraint that knowing the value of some element x_j doesn't alter the probability $\Pr(x_i = W_i \cdot h + b_i + \text{noise})$. Given how we've defined the variable $\boldsymbol{h}$, this means that knowing noise_j provides no clues about noise_i. Mathematically, the noise must have a diagonal covariance matrix.

- **Thoughts**. Not really seeing why this is useful/noteworthy. Feels very contrived (many assumptions) and restrictive – it only applies if the dependencies between each x_i can be modeled with a random variable $\boldsymbol{h}$ sampled from a unit variance Gaussian.

- **Probabilistic PCA**: Just factor analysis with $\boldsymbol{\psi} = \sigma^2 \boldsymbol{I}$. So zero-mean spherical Gaussian noise. It becomes regular PCA as $\sigma \to 0$. Here we can use an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$.

Autoencoders (Ch. 14)

 Written by Brandon McKinzie

Introduction. An autoencoder learns to copy its input to its output, via an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder function $\boldsymbol{r} = g(\boldsymbol{h})$. Modern autoencoders generalize this to allow for stochastic mappings $p_{encoder}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{decoder}(\boldsymbol{x} \mid \boldsymbol{h})$.

$\boldsymbol{r}$ for "reconstruction"

Undercomplete Autoencoders. Constrain dimension of $\boldsymbol{h}$ to be smaller than that of $\boldsymbol{x}$. The learning process minimizes some $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$, where the loss function could be e.g. mean squared error. Be careful not to have too many learnable parameters in the functions g and f (thus increasing model capacity), since that defeats the purpose of using an undercomplete autoencoder in the first place.

Regularized Autoencoders. We can remove the undercomplete constraint/necessity by modifying our loss function. For example, a **sparse autoencoder** one that adds a penalty $\Omega(\boldsymbol{h})$ to the loss function that encourages the *activations on* (not connections to/from) the hidden layer to be sparse. One way to achieve *actual zeros* in $\boldsymbol{h}$ is to use rectified linear units for the activations.

Greedy Layer-Wise Unsupervised Pretraining. Given:

- Unsupervised learning algorithm $\mathcal{L}$ which accepts as input a training set of examples $\boldsymbol{X}$, and outputs an encoder/feature function f.
- $f^{(i)}(\tilde{\boldsymbol{X}})$ denotes the output of the ith layer of f, given as *immediate input* the (possibly transformed) set of examples $\tilde{\boldsymbol{X}}$.
- Let m denote the number of layers ("stages") in the encoder function (note that each layer/stage here *must* use a representation learning algorithm for its $\mathcal{L}$ e.g. an RBM, autoencoder, sparse coding model, etc.)

The procedure is as follows:

1. Initialize.

$$f(\cdot) \leftarrow I(\cdot) \tag{103}$$
$$\tilde{\boldsymbol{X}} = \boldsymbol{X} \tag{104}$$

2. For each layer (stage) i in range(m), do:

$$f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}}) \tag{105}$$
$$f(\cdot) \leftarrow f^{(k)}\left(f\left(\cdot\right)\right) \tag{106}$$
$$\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}}) \tag{107}$$

   In English: just apply the regular learning/training process for each layer/stage **sequentially and individually**[33].

When this is complete, we can run **fine-tuning**: train all layers together (including any later layers that could not be pretrained) with a supervised learning algorithm. Note that we do indeed allow the pretrained encoding stages to be optimized here (i.e. not fixed).

[33] In other words, you proceed one layer at a time *in order*. You don't touch layer i until the weights in layer $i-1$ have been learned.

Structured Probabilistic Models for DL (Ch. 16)

Written by Brandon McKinzie

Motivation. In addition to classification, we can ask probabilistic models to perform other tasks such as density estimation ($\boldsymbol{x} \to p(\boldsymbol{x})$), denoising, missing value imputation, or sampling. What these [other] tasks have in common is they require a *complete understanding of the input*. Let's start with the most naive approach of modeling $p(\boldsymbol{x})$, where $\boldsymbol{x}$ contains n elements, each of which can take on k distinct values: we store a lookup table of all possible $\boldsymbol{x}$ and the corresponding probability value $p(\boldsymbol{x})$. This requires k^n parameters[34]. Instead, we use graphs to describe model structure (direct/indirect interactions) to drastically reduce the number of parameters.

Directed Models. Also called **belief networks** or **Bayesian networks**. Formally, a directed graphical model defined on a set of variables $\{\boldsymbol{x}\}$ is defined by a DAG, $\mathcal{G}$, whose vertices are the random variables in the model, and a set of **local conditional probability distributions**, $p(x_i \mid Pa_{\mathcal{G}}(x_i))$, where $Pa_{\mathcal{G}}(x_i)$ gives the parents of x_i in $\mathcal{G}$. The probability distribution over $\boldsymbol{x}$ is given by

$$p(\boldsymbol{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)) \tag{108}$$

Undirected Graphical Models. Also called **Markov Random Fields (MRFs)** or **Markov Networks**. Appropriate for situations where interactions do not have a well-defined direction. Each **clique** $\mathcal{C}$ (any set of nodes that are all [maximally] connected) in $\mathcal{G}$ is associated with a factor $\phi(\mathcal{C})$. The factor $\phi(\mathcal{C})$, also called a **clique potential**, is just a function (not necessarily a probability) that outputs a number when given a possible set of values over the nodes in $\mathcal{C}$. The output number measures the affinity of the variables in that clique for being in the states specified by the inputs. The set of all factors in $\mathcal{G}$ defines an **unnormalized probability distribution**:

Clique potentials are constrained to be nonnegative.

$$\widetilde{p}(\boldsymbol{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}) \tag{109}$$

[34] Consider the common NLP case where our vector $\boldsymbol{x}$ contains n word tokens, each of which can take on any symbol in our vocabulary of size v. If we assign $n = 100$ and $v = 100,000$, which are relatively common values for this case, this amounts to $(1e5)^{1e2} = 10^{500}$ parameters.

The Partition Function. To obtain a valid probability distribution, we must normalize the probability distribution:

$$p(\boldsymbol{x}) = \frac{1}{Z}\widetilde{p}(\boldsymbol{x}) \tag{110}$$

$$Z = \int \widetilde{p}(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{111}$$

where the normalizing function $Z = Z(\{\phi\})$ is known as the **partition function** (physicz sh0ut0uT). It is typically intractable to compute, so we resort to approximations. Note that Z isn't even guaranteed to exist – it's only for those definitions of the clique potentials that cause the integral over $\widetilde{p}(\boldsymbol{x})$ to converge/be defined.

Energy-Based Models (EBMs). A convenient way to enforce $\forall \boldsymbol{x},\ \widetilde{p}(\boldsymbol{x}) > 0$ is to use EBMs, where

$$\widetilde{p}(\boldsymbol{x}) \triangleq \exp\left(-E(\boldsymbol{x})\right) \tag{112}$$

and $E(\boldsymbol{x})$ is known as the **energy function**[35]. Many algorithms need to compute not $p_{model}(\boldsymbol{x})$ but only $\log \widetilde{p}_{model}(\boldsymbol{x})$ (unnormalized log probabilities - logits!). For EBMs with latent variables $\boldsymbol{h}$, such algorithms are phrased in terms of the **free energy**:

$$\mathcal{F}(\boldsymbol{x} = x) = -\log \sum_h \exp\left(-E(\boldsymbol{x} = x,\ \boldsymbol{h} = h)\right) \tag{113}$$

where we sum over all possible assignments of the latent variables.

Separation and D-Separation. We want to know which subsets of variables are conditionally independent from each other, given the values of other subsets of variables. A set of variables $\mathbb{A}$ is **separated** (if undirected model)/**d-separated** (if directed model) from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$.

- **Separation**. For *undirected* models. If variables a and b are connected by a path involving only <u>unobserved</u> variables (an **active** path), then a and b are *not* separated. Otherwise, they are separated. Any paths containing at least one observed variable are called **inactive**.
- **D-Separation**[36]. For *directed* models. Although there are rules that help determine whether a path between a and b is d-separated, it is simplest to just determine whether a is independent from b given any observed variables along the path.

[35] Physics throwback: this mirrors the Boltzmann factor, $\exp(-\varepsilon/\tau)$, which is proportional to the probability of the system being in quantum energy state ε.

[36] The D stands for dependence.

Sampling from Graphical Models

For directed graphical models, we can do **ancestral sampling** to produce a sample $\boldsymbol{x}$ from the joint distribution represented by the model. Just sort the variables x_i into a topological ordering such that $\forall i, j : j > i \iff x_i \in Pa_{\mathcal{G}}(x_j)$. To produce the sample, just sequentially sample from the beginning, $x_1 \sim P(x_1)$, $x_2 \sim P(x_2 \mid Pa_{\mathcal{G}}(x_1))$, etc.

For undirected graphical models, one simple approach is **Gibbs sampling**. Essentially, this involves drawing a conditioned sample from $x_i \sim p(x_i \mid \text{neighbors}(x_i))$ for each x_i. This process is repeated many times, where each subsequent pass uses the previously sampled values in neighbors(x_i) to obtain an asymptotically converging [to the correct distribution] estimate for a sample from $p(\boldsymbol{x})$.

Inference and Approximate Inference

One of the main tasks with graphical models is predicting the values of some subset of variables given another subset: inference. Although the graph structures we've discussed allow us to represent complicated, high-dimensional distributions with a reasonable number of parameters, the graphs used for deep learning are usually not restrictive enough to allow efficient inference. **Approximate inference** for deep learning usually refers to variational inference, in which we approximate the distribution $p(\boldsymbol{h} \mid \boldsymbol{v})$ by seeking an approximate distribution $q(\boldsymbol{h} \mid \boldsymbol{v})$ that is as close to the true one as possible.

Example: Restricted Boltzmann Machine. The quintessential example of how graphical models are used for deep learning. The canonical RBM is an energy-based model with **binary** visible and hidden units. Its energy function is

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^T \boldsymbol{v} - \boldsymbol{c}^T \boldsymbol{h} - \boldsymbol{v}^T \boldsymbol{W} \boldsymbol{h} \tag{114}$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. One could interpret the values of the bias parameters as the affinities for the associated variable being its given value, and the value $\boldsymbol{W}_{i,j}$ as the affinity of v_i being its value and h_j being its value at the same time[37].

The restrictions on the RBM structure, namely the fact that there are no intra-layer connections, yields nice properties. Since $\widetilde{p}(\boldsymbol{h}, \boldsymbol{v})$ can be factored into clique potentials, we can say

[37] More concretely, remember that $\boldsymbol{v}$ is a one-hot vector representing some state that can assume len$(\boldsymbol{v})$ unique values, and similarly for $\boldsymbol{h}$. Then $\boldsymbol{W}_{i,j}$ gives the affinity for the state associated with v being its ith value *and* the state associated with h being its jth value.

that:

$$p(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i p(h_i \mid \boldsymbol{v}) \tag{115}$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_i p(v_i \mid \boldsymbol{h}) \tag{116}$$

Also, due to the restriction of binary variables, each of the conditionals is easy to compute, and can be quickly derived as

$$p(h_i = 1 \mid \boldsymbol{v}) = \sigma\left(c_i + \boldsymbol{v}^T \boldsymbol{W}_{:,i}\right) \tag{117}$$

allowing for efficient block Gibbs sampling.

Monte Carlo Methods (Ch. 17)

 Written by Brandon McKinzie

Monte Carlo Sampling (Basics). We can approximate the value of a (usually prohibitively large) sum/integral by viewing it as an *expectation* under some distribution. We can then approximate its value by taking samples from the corresponding probability distribution and taking an empirical average. Mathematically, the basic idea is show below:

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_p \left[f(\mathbf{x}) \right] \quad \rightarrow \quad \hat{s}_n = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{118}$$

As we've seen before, the empirical average is an unbiased[38] estimator. Furthermore, the central limit theorem tells us that the distribution of $\hat{s}_n$ converges to a normal distribution with mean s and variance $\mathrm{Var}\left[f(\boldsymbol{x}) \right] / n$.

Importance Sampling. What if it's not feasible for us to sample from p? We can approach this a couple ways, both of which will exploit the following identity:

$$p(\boldsymbol{x}) f(\boldsymbol{x}) = q(\boldsymbol{x}) \frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{122}$$

- **Optimal importance sampling**. We can use the aforementioned identity/decomposition to find the **optimal** $q*$ – optimal in terms of number of samples required to achieve a given level of accuracy. First, we rewrite our estimator $\hat{s}_p$ (they now use subscript to denote the sampling distribution) as $\hat{s}_q$:

$$\hat{s}_q = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} \tag{123}$$

[38] Recall that expectations on such an average are still taken over the underlying (assumed) probability distribution:

$$\mathbb{E}_p \left[\hat{s}_n \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right] \tag{119}$$

$$= \frac{1}{n} \sum_{i=1}^{n} s \tag{120}$$

$$= s \tag{121}$$

You should think of the expectation $\mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right]$ as the expected value of the *random sample* from the underlying distribution, which of course is s, because we defined it that way.

At first glance, it feels a little wonky, but recognize that we are *sampling from q instead of p* (i.e. if this were an integral, it would be over $q(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$). The catch is that, now, the variance can be greatly sensitive to the choice of q:

$$\mathrm{Var}\left[\hat{s}_q\right] = \mathrm{Var}\left[\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}\right]/n \tag{124}$$

with the optimal (minimum) value of q at:

$$q* = \frac{p(\boldsymbol{x}) \mid f(\boldsymbol{x}) \mid}{Z} \tag{125}$$

- **Biased importance sampling**. Computing the optimal value of q can be as challenging/infeasible as sampling from p. Biased sampling does not require us to find a normalization constant for p or q. Instead, we compute:

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{126}$$

where $\tilde{p}$ and $\tilde{q}$ are the unnormalized forms of p and q, and the $\boldsymbol{x}^{(i)}$ samples are still drawn from [the original/unknown] q. $\mathbb{E}\left[\hat{s}_{BIS}\right] \neq s$ except asymptotically when $n \to \infty$.

Noise Contrastive Estimation (NCE) (18.6). We now estimate

$$\log p_{model}(\boldsymbol{x}) = \log \tilde{p}_{model}(\boldsymbol{x}; \boldsymbol{\theta}) + c \tag{127}$$

and explicitly learn an approximation, c, for $-\log Z(\boldsymbol{\theta})$. Obviously MLE would just try jacking up c to maximize this, so we adopt a surrogate supervised training problem: binary classification that a given sample $\boldsymbol{x}$ belongs to the (true) data distribution p_{data} or to the noise distribution p_{noise}. We introduce binary variable y to indicate whether the sample is in the true data distribution (y=1) or the noise distribution (y=0). Our surrogate model is thus defined by

$$p_{joint}(y{=}1) = \frac{1}{2} \tag{128}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}1) = p_{model}(\boldsymbol{x}) \tag{129}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}0) = p_{noise}(\boldsymbol{x}) \tag{130}$$

We can now use MLE on the optimization problem,

$$\boldsymbol{\theta}, c = \arg\max_{\boldsymbol{\theta}, c} \mathbb{E}_{\boldsymbol{x}, y \sim p_{train}} \left[\log p_{joint}(y \mid \boldsymbol{x})\right] \tag{131}$$

$$p_{joint}(y{=}1 \mid \boldsymbol{x}) = \frac{p_{model}(\boldsymbol{x})}{p_{model}(\boldsymbol{x}) + p_{noise}(\boldsymbol{x})} \tag{132}$$

$$= \frac{1}{1 + p_{noise}(\boldsymbol{x})/p_{model}(\boldsymbol{x})} \tag{133}$$

$$= \sigma\left(\log p_{model}\left(\boldsymbol{x}\right) - \log p_{noise}\left(\boldsymbol{x}\right)\right) \tag{134}$$

Approximate Inference (Ch. 19)

Written by Brandon McKinzie

Overview. Most graphical models with multiple layers of hidden variables have intractable posterior distributions. This is typically because the partition function scales exponentially with the number of units and/or due to marginalizing out latent variables. Many approximate inference approaches make use of the observation that exact inference can be described as an optimization problem.

Assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We want to compute $\log p(\boldsymbol{v};\boldsymbol{\theta})$, but it's too costly to marginalize out $\boldsymbol{h}$. Instead, we compute a lower bound $\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q)$ – often called the **evidence lower bound** (ELBO) or negative **variational free energy** – on $\log p(\boldsymbol{v};\boldsymbol{\theta})$[39]:

$$\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q) = \log p(\boldsymbol{v};\boldsymbol{\theta}) - D_{KL}\left(q(\boldsymbol{h} \mid \boldsymbol{v}) \mid\mid p(\boldsymbol{h} \mid \boldsymbol{v};\boldsymbol{\theta})\right) \tag{135}$$

$$= -\mathbb{E}_{\boldsymbol{h}\sim q(\boldsymbol{h}|\boldsymbol{v})}\left[\log p(\boldsymbol{h},\boldsymbol{v})\right] + H(q(\boldsymbol{h} \mid \boldsymbol{v})) \tag{136}$$

q is an arbitrary probability distribution over $\boldsymbol{h}$. Note that the book will write q when they really mean $q(\boldsymbol{h} \mid \boldsymbol{v})$.

where the second form is the more canonical definition[40]. Note that $\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q)$ is a lower-bound on $\log p(\boldsymbol{v};\boldsymbol{\theta})$ by definition, since

$$\log p(\boldsymbol{v};\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q) = D_{KL}\left(q(\boldsymbol{h} \mid \boldsymbol{v})||p(\boldsymbol{h} \mid \boldsymbol{v};\boldsymbol{\theta})\right) \geq 0$$

With equality (to zero) iff q is the same distribution as $p(\boldsymbol{h} \mid \boldsymbol{v})$. In other words, $\mathcal{L}$ can be viewed as a function parameterized by q that's maximized when q is $p(\boldsymbol{h} \mid \boldsymbol{v})$, and with maximal value $\log p(\boldsymbol{v})$. Therefore, we can cast the *inference* problem of computing the (log) probability of the observed data $\log p(\boldsymbol{v})$ into an *optimization* problem of maximizing $\mathcal{L}$. Exact inference can be done by searching over a family of functions that contains $p(\boldsymbol{h} \mid \boldsymbol{v})$.

[39] Recall that $D_{KL}(P||Q) = \mathbb{E}_{x\sim P(x)}\left[\log \frac{P(x)}{Q(x)}\right]$

[40] This can be derived easily from the first form. Hint:

$$\log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{p(\boldsymbol{h} \mid \boldsymbol{v})} = \log \frac{q(\boldsymbol{h} \mid \boldsymbol{v})}{p(\boldsymbol{h},\boldsymbol{v};\boldsymbol{\theta})/p(\boldsymbol{v};\boldsymbol{\theta})}$$

Expectation Maximization (19.2). Technically not an approach to approximate inference, but rather an approach to learning with an approximate posterior. Popular for training models with latent variables. The EM algorithm consists of alternating between the followi[p]ng 2 steps until convergence:

1. **E-step**. For each training example $\boldsymbol{v}^{(i)}$ (in current batch or full set), set

$$q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}) = p(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)}) \tag{137}$$

   where $\boldsymbol{\theta}^{(0)}$ denotes the current parameter values of the model at the beginning of the E-step. This can also be interpreted as maximizing $\mathcal{L}$ w.r.t. q.
2. **M-step**. Update the parameters $\boldsymbol{\theta}$ by completely or partially finding

$$\arg\max_{\boldsymbol{\theta}} \sum_i \mathcal{L}\left(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}, q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)})\right) \tag{138}$$

Boltzmann Machines (20.1). An energy-based model over a d-dimensional binary random vector $\mathbf{x} \in \{0,1\}^d$. The energy function is simply $E(\boldsymbol{x}) = -\boldsymbol{x}^T\boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$, i.e. parameters between all pairs of x_i, x_j, and bias parameters for each x_i[41]. In settings where our data consists of samples of fully observed $\boldsymbol{x}$, this is clearly limited to very simple cases, since e.g. the probability of some x_i being on is given by logistic regression from the values of the other units.

Proof: prob of x_i being on is logistic regression on other units

It's important to be as specific as possible here, since the task stated as-is is ambiguous. We want to prove that the probability of some fully observed state $\boldsymbol{x}$ that has its ith element clamped to 1, which I'll denote as $p_{i=on}(\boldsymbol{x})$, is logistic regression over the other units.

To prove this, it's easier to use the conventional definition where $\boldsymbol{U}$ is symmetric with zero diagonal, and we write $E(\boldsymbol{x})$ as

$$E(\boldsymbol{x}) = -\sum_{i=1}^{d}\sum_{j=i+1}^{d} x_i U_{i,j} x_j - \sum_{i=1}^{d} b_i x_i \tag{139}$$

where the difference is that we explicitly only sum over the upper triangle of $\boldsymbol{U}$.

Intuitively, since $p(\{\boldsymbol{x}\}_{j\neq i}) = p_{i=on}(\boldsymbol{x}) + p_{i=off}(\boldsymbol{x})$, our final formula for $p_{i=on}$ should only contain terms involving the parameters that interact with x_i, and only for those cases where $x_i = 1$. This motivates exploring the formula for $\Delta E_i(\boldsymbol{x}) \triangleq E_{i=off} - E_{i=on}$ where I've dropped the explicit notation on $\boldsymbol{x}$ for simplicity/space. Before jumping in to deriving this, step back and realize that ΔE_i will only contain summation terms where either the row or column index of $\boldsymbol{U}$ is i, and only for terms with bias element b_i. Since our summation is over the upper triangle of $\boldsymbol{U}$, this means terms along the slices $U_{i,i+1:d}$ and $U_{1:i-1,i}$. Now there is no derivation needed and we can simply write

$$\Delta E_i = \sum_{k=i+1}^{d} U_{i,k} x_k + \sum_{k=1}^{i-1} x_k U_{k,i} + b_i \tag{140}$$

The goal is to use this to get a logistic-regression-like formula for $p_{i=on}$, so we should now think about the relationship between any given $p(\boldsymbol{x})$ and the associated $E(\boldsymbol{x})$. The critical observation is that $E(\boldsymbol{x}) = -\ln p(\boldsymbol{x}) - \ln Z$, which therefore means

$$\Delta E_i = \ln p_{i=on}(\boldsymbol{x}) - \ln p_{i=off}(\boldsymbol{x}) = -\ln\left(\frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})}\right) \tag{141}$$

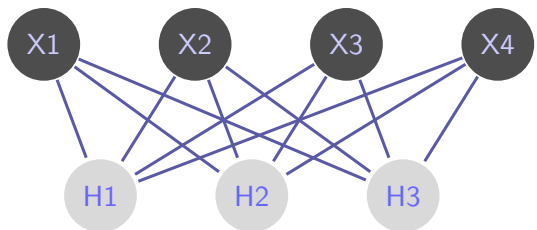
$$\exp(-\Delta E_i) = \frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})} = \frac{1}{p_{i=on}(\boldsymbol{x})} - 1 \tag{142}$$

$$\therefore p_{i=on}(\boldsymbol{x}) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{143}$$

Since ΔE_i is a linear function of all other units, we have proven that $p_{i=on}(\boldsymbol{x})$ for some state $\boldsymbol{x}$ reduces to logistic regression over the other units.

[41] Authors are being lazy because it's assumed the reader is familiar (which is fair, I guess). i.e. they aren't mentioning that this formula implies that $\boldsymbol{U}$ is either lower or upper triangular, and the diagonal is zero.

Restricted Boltzmann Machines (20.2). A BM with variables partitioned into two sets: hidden and observed. The graphical model is bipartite over the hidden and observed nodes, as I've drawn in the example below.



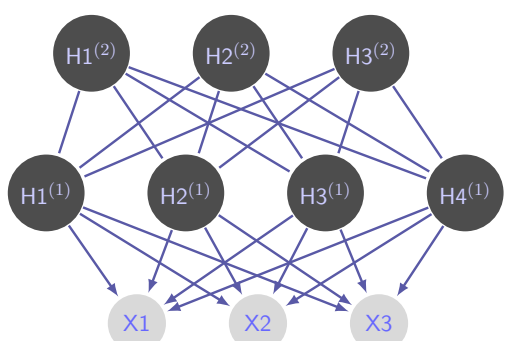
Although the joint distribution $p(\boldsymbol{x}, \boldsymbol{h})$ has a potentially intractable partition function, the conditional distributions can be computed efficiently by exploiting independencies:

$$p(\boldsymbol{h} \mid \boldsymbol{x}) = \prod_{j=1}^{n_h} \sigma\left([2\boldsymbol{h} - 1] \odot [\boldsymbol{c} + \boldsymbol{W}^T \boldsymbol{x}]\right)_j \tag{144}$$

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_{i=1}^{n_x} \sigma\left([2\boldsymbol{x} - 1] \odot [\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}]\right)_i \tag{145}$$

where $\boldsymbol{b}$ and $\boldsymbol{c}$ are the observed and hidden bias parameters, respectively.

Deep Belief Networks (20.3). Several layers of (usually binary) latent variables and a single observed layer. The "deepest" (away from the observed) layer connections are undirected, and all other layers are directed and pointing toward the data. I've drawn an example below.



We can sample from a DBN via first Gibbs sampling on the undirected layer, then ancestral sampling through the rest of the (directed) model to eventually obtain a sample from the visible units.

Deep Boltzmann Machines (20.4). Same as DBNs, but now all layers are undirected. Note that this is very close to the standard RBM, since we have a set of hidden and observed variables, except now we interpret certain subgroups of hidden units as being in a "layer", thus allowing for connections between hidden units in adjacent layers. What's interesting is that this still defines a bipartite graph, with odd-numbered layers on one side and even on the other[42].

Differentiable Generator Networks (20.10.2). Use a differentiable function $g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$ to transform samples of latent variables $\mathbf{z}$ to either (a) samples $\mathbf{x}$, or (b) distributions over samples $\mathbf{x}$. For an example of case (a), the standard procedure for sampling from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is to first sample from $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ into a generator network consisting of a single affine layer:

Case a: interpret $g(\boldsymbol{z})$ as emitting $\boldsymbol{x}$ directly.

$$\boldsymbol{x} \leftarrow g(\boldsymbol{z}) = \boldsymbol{\mu} + \boldsymbol{L}\boldsymbol{z}$$

where $\boldsymbol{L}$ is the *Cholesky decomposition*[43] of $\boldsymbol{\Sigma}$. In general, we think of the generator function g as providing a change of variables that transforms the distribution over $\mathbf{z}$ into the desired distribution $\mathbf{x}$. Of course, there *is* an exact formula for doing this,

$$p_x(\boldsymbol{x}) = \frac{p_z(g^{-1}(\boldsymbol{x}))}{\left|\det \frac{\partial g}{\partial \boldsymbol{z}}\right|} \tag{146}$$

but it's usually far easier to use indirect means of learning g, rather than trying to maximize/evaluation $p_x(\boldsymbol{x})$ directly.

For case (b), the common approach is to train the generator net to emit conditional probabilities

Case b: interpret $g(\boldsymbol{z})$ as emitting $p(\boldsymbol{x} \mid \boldsymbol{z})$.

$$p(x_i \mid \boldsymbol{z}) = g(\boldsymbol{z})_i \qquad\qquad p(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}}\left[p(\boldsymbol{x} \mid \boldsymbol{z})\right] \tag{147}$$

which can also support generating discrete data (case a cannot). The challenge in training generator networks is that we often have a set of examples $\boldsymbol{x}$, but the value of $\boldsymbol{z}$ for each $\boldsymbol{x}$ is not fixed and known ahead of time. We'll now look at some ways of training generator nets given only training samples for $\boldsymbol{x}$. Note that such a setting is very unlike unsupervised learning, where we typically interpret $\boldsymbol{x}$ as inputs that we don't have labels for, while here we interpret $\boldsymbol{x}$ as *outputs* that we don't know the associated inputs for.

[42] Recall that this immediately implies that units in all odd layers are conditionally independent given the even layers (and vice-versa for even to odd).

[43] The [unique] Cholesky decomposition of a (real-symmetric) p.d. matrix $\boldsymbol{A}$ is a decomposition of the form $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is lower triangular.

Variational Autoencoders (20.10.3). VAEs are directed models that use learned approximate inference and can be trained purely with gradient-based methods. To generate a sample $\boldsymbol{x}$, the VAE first samples $\boldsymbol{z}$ from the *code distribution* $p_{model}(\boldsymbol{z})$. This sample is then fed through the a differentiable generator network $g(\boldsymbol{z})$. Finally, $\boldsymbol{x}$ is sampled from $p_{model}(\boldsymbol{x}; g(\boldsymbol{z})) = p_{model}(\boldsymbol{x} \mid \boldsymbol{z})$.

Papers and Tutorials

Contents

4.1 WaveNet 69

4.2 Neural Style 73

4.3 Neural Conversation Model 75

4.4 NMT By Jointly Learning to Align & Translate 77

4.4.1 Detailed Model Architecture 78

4.5 Effective Approaches to Attention-Based NMT 80

4.6 Using Large Vocabularies for NMT 82

4.7 Candidate Sampling – TensorFlow 85

4.8 Attention Terminology 87

4.9 TextRank 89

4.9.1 Keyword Extraction 91

4.9.2 Sentence Extraction 92

4.10 Simple Baseline for Sentence Embeddings 93

4.11 Survey of Text Clustering Algorithms 95

4.11.1 Feature Selection and Transformation Methods 95

4.11.2 Distance-based Clustering Algorithms 98

4.11.3 Probabilistic Document Clustering and Topic Models 99

4.11.4 Online Clustering with Text Streams 101

4.11.5 Semi-Supervised Clustering 102

4.12 Deep Sentence Embedding Using LSTMs 103

4.13 Clustering Massive Text Streams 106

4.14 Supervised Universal Sentence Representations (InferSent) 108
4.15 Dist. Rep. of Sentences from Unlabeled Data (FastSent) 109
4.16 Latent Dirichlet Allocation 111
4.17 Conditional Random Fields 114
4.18 Attention Is All You Need 117
4.19 Hierarchical Attention Networks 120
4.20 Joint Event Extraction via RNNs 123
4.21 Event Extraction via Bidi-LSTM Tensor NNs 125
4.22 Reasoning with Neural Tensor Networks 127
4.23 Language to Logical Form with Neural Attention 128
4.24 Seq2SQL: Generating Structured Queries from NL using RL 130
4.25 SLING: A Framework for Frame Semantic Parsing 133
4.26 Poincaré Embeddings for Learning Hierarchical Representations 135
4.27 Enriching Word Vectors with Subword Information (FastText) 137
4.28 DeepWalk: Online Learning of Social Representations 139
4.29 Review of Relational Machine Learning for Knowledge Graphs 141
4.30 Fast Top-K Search in Knowledge Graphs 144
4.31 Dynamic Recurrent Acyclic Graphical Neural Networks (DRAGNN) 146
4.31.1 More Detail: Arc-Standard Transition System 149
4.32 Neural Architecture Search with Reinforcement Learning 150
4.33 Joint Extraction of Events and Entities within a Document Context 152
4.34 Globally Normalized Transition-Based Neural Networks 155
4.35 An Introduction to Conditional Random Fields 158
4.35.1 Inference (Sec. 4) 162
4.35.2 Parameter Estimation (Sec. 5) 165
4.35.3 Related Work and Future Directions (Sec. 6) 168
4.36 Co-sampling: Training Robust Networks for Extremely Noisy Supervision . . . 169
4.37 Hidden-Unit Conditional Random Fields 170
4.37.1 Detailed Derivations 172

4.38 Pre-training of Hidden-Unit CRFs 177
4.39 Structured Attention Networks 179
4.40 Neural Conditional Random Fields 181
4.41 Bidirectional LSTM-CRF Models for Sequence Tagging 183
4.42 Relation Extraction: A Survey 184
4.43 Neural Relation Extraction with Selective Attention over Instances 187
4.44 On Herding and the Perceptron Cycling Theorem 189
4.45 Non-Convex Optimization for Machine Learning 191
4.45.1 Non-Convex Projected Gradient Descent (3) 194
4.46 Improving Language Understanding by Generative Pre-Training 195
4.47 Deep Contextualized Word Representations 196
4.48 Exploring the Limits of Language Modeling 198

WaveNet

Written by Brandon McKinzie

Abstract.

- **What's WaveNet?** A deep neural network for generating raw audio waveforms.
- **How does it generate them? IDK**
- **What's it good for?** Text-to-speech, generating music, any waveform really.

Introduction.

- Inspired by recent advances in **neural autoregressive generative models**, and based on the PixelCNN architecture.
- Long-range dependencies dealt with via "dilated causal convolutions, which exhibit very large receptive fields."

WaveNet. The joint probability of a waveform $x = \{x_1, \ldots, x_T\}$ is factorised as a product of conditional probabilities,

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \tag{148}$$

which are modeled by a stack of convolutional layers (no pooling).

> The model outputs a categorical distribution over the next value xt with a softmax layer and it is optimized to maximize the log-likelihood of the data w.r.t. the parameters.

Main ingredient of WaveNet is *dilated* causal convolutions, illustrated below. Note the absence of recurrent connections, which makes them faster to train than RNNs, but at the cost of requiring many layers, or large filters to increase the receptive field[44].

[44] Loose interpretation of receptive fields here is that large fields can take into account more info (back in time) as opposed to smaller fields, which can be said to be "short-sighted"

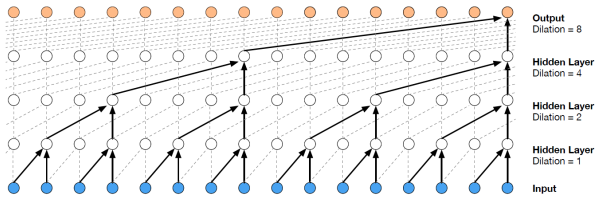


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Excellent concise definition from paper:

> A dilated convolution (a convolution with holes) is a convolution where the filter is applied over an area larger than its length by skipping input values with a certain step. It is equivalent to a convolution with a larger filter derived from the original filter by dilating it with zeros, but is significantly more efficient. A dilated convolution effectively allows the network to operate on a coarser scale than with a normal convolution. This is similar to pooling or strided convolutions, but here the output has the same size as the input. As a special case, dilated convolution with dilation 1 yields the standard convolution.

Softmax distributions. Chose to model the conditional distributions $p(x_t \mid x_1, \ldots, x_{t-1})$ with a softmax layer. To deal with the fact that there are 2^{16} possible values, first apply a "μ-law companding transformation" to data, and then quantize it to 256 possible values:

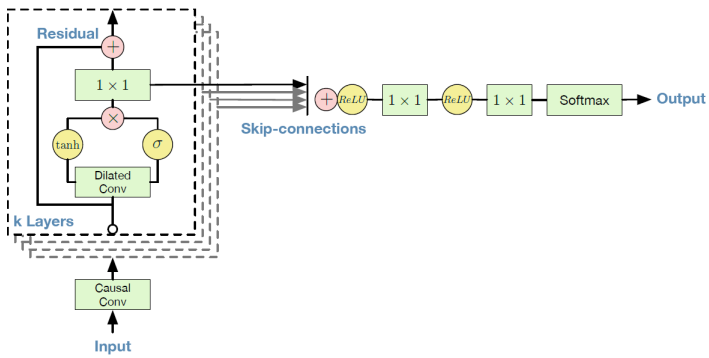
$$f(x_t) = \text{sign}(x_t) \frac{\ln(1 + \mu|x_t|)}{\ln(1 + \mu)} \tag{149}$$

which (after plotting in Wolfram) looks identical to the sigmoid function.

Gated activation and res/skip connections. Use the same gated activation unit as PixelCNN:

$$z = \tanh\left(W_{f,k} * x\right) \ \odot \ \sigma\left(W_{g,k} * x\right) \tag{150}$$

where $*$ denotes conv operator, $\odot$ denotes elem-wise mult., k is layer index, f, g denote filter/-gate, and W is learnable conv filter. This is illustrated below, along with the residual/skip connections used to speed up convergence/enable training deeper models.



Conditional Wavenets. Can also model conditional distribution of x given some additional h (e.g. speaker identity).

$$p(x \mid h) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, h) \tag{151}$$

→ **Global conditioning**. Single h that influences output dist. accross all times. Activation becomes:

$$z = \tanh\left(W_{f,k} * x + V_{f,k}^T h\right) \ \odot \ \sigma\left(W_{g,k} * x + V_{g,k}^T h\right) \tag{152}$$

→ **Local conditioning** (confusing). Have a second time-series h_t. They first transform this h_t using a "transposed conv net (learned unsampling) that maps it to a new time-series $y = f(h)$ w/same resolution as x.

Experiments.

- **Multi-Speaker Speech Generation**. Dataset: multi-speaker corpus of 44 hours of data from 109 different speakers[45]. Receptive field of 300 milliseconds.
- **Text-to-Speech**. Single-speaker datasets of 24.6 hours (English) and 34.8 hours (Chinese) speech. Locally conditioned on *linguistic features*. Receptive field of 240 milliseconds. Outperformed both LSTM-RNN and HMM.
- **Music**. Trained the WaveNets to model two music datasets: (1) 200 hours of annotated music audio, and (2) 60 hours of solo piano music from youtube. Larger receptive fields sounded more musical.
- **Speech Recognition**. "With WaveNets we have shown that layers of dilated convolutions allow the receptive field to grow longer in a much cheaper way than using LSTM units."

[45] Speakers encoded as ID in form of a one-hot vector

Conclusion (verbatim): "This paper has presented WaveNet, a deep generative model of audio data that operates directly at the waveform level. WaveNets are autoregressive and combine causal filters with dilated convolutions to allow their receptive fields to grow exponentially with depth, which is important to model the long-range temporal dependencies in audio signals. We have shown how WaveNets can be conditioned on other inputs in a global (e.g. speaker identity) or local way (e.g. linguistic features). When applied to TTS, WaveNets produced samples that outperform the current best TTS systems in subjective naturalness. Finally, WaveNets showed very promising results when applied to music audio modeling and speech recognition."

Neural Style

Written by Brandon McKinzie

Notation.

- **Content image**: $\boldsymbol{p}$
- **Filter responses**: the matrix $P^l \in \mathcal{R}^{N_l \times M_l}$ contains the activations of the filters in layer l, where P^l_{ij} would give the activation of the ith filter at position j in layer l. N_l is number of feature maps, each of size M_l (height $\times$ width of the feature map)[46].
- **Reconstructed image**: $\boldsymbol{x}$ (initially random noise). Denote its corresponding filter response matrix at layer l as P^l.

Content Reconstruction.

1. Feed in **content image** $\boldsymbol{p}$ into pre-trained network, saving any desired filter responses during the forward pass. These are interpreted as the various "encodings" of the image done by the network. Think of them analogously to "ground-truth" labels.

2. Define $\boldsymbol{x}$ as the **generated image**, which we first initialize to random noise. We will be changing the pixels of $\boldsymbol{x}$ via gradient descent updates.

3. Define the **loss function**. After each forward pass, evaluate with squared-error loss between the two representations at the layer of interest:

$$\mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}, l) = \frac{1}{2} \sum_{i,j} (F^l_{ij} - P^l_{ij})^2 \tag{1}$$

$$\frac{\partial \mathcal{L}_{content}}{\partial F^l_{ij}} = \begin{cases} (F^l - P^l)_{ij} & F^l_{ij} > 0 \\ 0 & F^l_{ij} < 0 \end{cases} \tag{2}$$

   where it appears we are assuming ReLU activations (**?**).

4. Compute iterative updates to $\boldsymbol{x}$ via **gradient descent** until it generates the same response in a certain layer of the CNN as the original image $\boldsymbol{p}$.

[46] If not clear, M_l is a scalar, for any given value of l.

Style Representation. On top of the CNN responses in each layer, the authors built a style representation that computes the correlations between the different [aforementioned] filter responses. The correlation matrix for layer l is denoted in the standard way with a Gram matrix $G^l \in \mathcal{R}^{N_l \times N_l}$, with entries

$$G^l_{ij} = \langle F^l_i, F^l_j \rangle = \sum_k F^l_{ik} F^l_{jk} \tag{3}$$

To generate a texture that matches the style of a given image, do the following.

1. Let $\boldsymbol{a}$ denote the original [style] image, with corresponding A^l. Let $\boldsymbol{x}$, initialized to random noise, denote the generated [style] image, with corresponding G^l.

2. The contribution to the loss of layer l, denoted E_l, to the total loss, denoted $\mathcal{L}_{style}$, is given by

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (G^l_{ij} - A^l_{ij})^2 \tag{4}$$

$$\mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) = \sum_{l=0}^{L} w_l E_l \tag{5}$$

$$\frac{\partial E_l}{\partial F^l_{ij}} = \begin{cases} \frac{1}{N_l^2 M_l^2} \left((F^l)^T (G^l - A^l) \right)_{ji} & F^l_{ij} > 0 \\ 0 & F^l_{ij} < 0 \end{cases} \tag{6}$$

   where w_l are [as of yet unspecified] weighting factors of the contribution of layer l to the total loss.

Mixing content with style. Essentially a joint minimization that combines the previous two main ideas.

1. Begin with the following images: white noise $\boldsymbol{x}$, content image $\boldsymbol{p}$, and style image $\boldsymbol{a}$.

2. The loss function to minimize is a linear combination of 1 and 5:

$$\mathcal{L}_{total}(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{x}, l) = \alpha \mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}, l) + \beta \mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) \tag{7}$$

   Note that we can choose which layers L_{style} uses by tweaking the layer weights w_l. For example, the authors chose to set $w_l = 1/5$ for 'conv[1, 2, 4, 5]_1' and 0 otherwise. For the ratio α/β, they explored 1×10^{-3} and 1×10^{-4}.

Papers and Tutorials **February 8**

Neural Conversation Model

Table of Contents Local *Written by Brandon McKinzie*

[Reminder: **red text** means I need to come back and explain what is meant, once I understand it.]

Abstract. This paper presents a simple approach for conversational modeling which uses the sequence to sequence framework. It can be trained end-to-end, meaning fewer hand-crafted rules. The **lack of consistency** is a common failure of our model.

Introduction. Major advantage of the seq2seq model is it requires little feature engineering and domain specificity. Here, the model is tested on chat sessions from an IT helpdesk dataset of conversations, as well as movie subtitles.

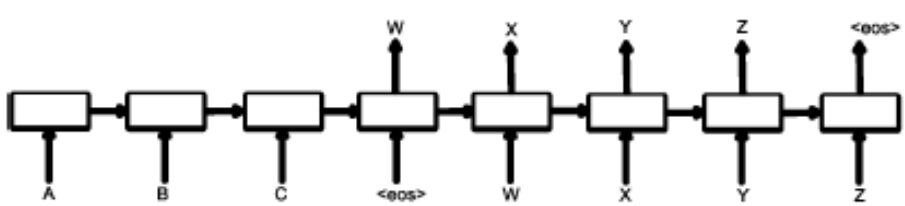
Related Work. The authors' approach is based on the following (linked and saved) papers on seq2seq:

- Kalchbrenner & Blunsom, 2013.
- Sutskever et al., 2014. (Describes Seq2Seq model)
- Bahdanau et al., 2014.

Model. Succinctly described by the authors:

> The model reads the input sequence one token at a time, and predicts the output sequence, also one token at a time. During training, the true output sequence is given to the model, so learning can be done by backpropagation. The model is trained to maximize the cross entropy of the correct sequence given its context. During inference, in which the true output sequence is not observed, we simply feed the predicted output token as input to predict the next output. This is a greedy inference approach.

Example of less greedy approach: **beam search**.



The **thought vector** is the hidden state of the model when it receives [as input] the end of sequence symbol $\langle eos \rangle$, because it stores the info of the sentence, or *thought*, "ABC". The authors acknowledge that this model will *not* be able to "solve" the problem of modeling dialogue due to the objective function not capturing the actual objective achieved through human communication, which is typically longer term and based on exchange of information [rather than next step prediction][47].

Ponder: what *would* be a reasonable objective function & model for conversation?

IT Data & Experiment.

Reminder: Check out this git repo

- **Data Description**: Customers talking to IT support, where typical interactions are 400 words long and turn-taking is clearly signaled.
- **Training Data**: 30M tokens, 3M of which are used as validation. They built a vocabulary of the most common 20K words, and introduced special tokens indicating turn-taking and actor.
- **Model**: A single-layer LSTM with 1024 memory cells.
- **Optimization**: SGD with gradient clipping.
- **Perplexity**: At convergence, achieved **perplexity** of 8, whereas an n-gram model achieved 18.

[47] I'd imagine that, in order to model human conversation, one obvious element needed would be a *memory*. Reminds me of DeepMind's DNC. There would need to be some online filtering & output process to capture the crucial aspects/info to store in memory for later, and also some method of retrieving them when needed later. The method for retrieval would likely be some inference process where, given a sequence of inputs, the probability of them being related to some portion of memory could be trained. This would allow for conversations that stretch arbitrarily back in the past. Also, when storing the memories, I'd imagine a reasonable architecture would be some encoder-decoder for a sparse distributed representation of memory.

NMT By Jointly Learning to Align & Translate

Written by Brandon McKinzie

[Bahdanau et. al, 2014]. The primary motivation for me writing this is to better understand the **attention mechanism** in my sequence to sequence chatbot implementation.

Abstract. The authors claim that using a fixed-length vector [in the vanilla encoder-decoder for NMT] is a bottleneck. They propose allowing a model to (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly.

Learning to Align[48] and translate.

- **Decoder**. Their encoder defines the conditional output distribution as

$$p(y_i \mid y_1, \ldots, y_{i-1}, \boldsymbol{x}) = g(y_{i-1}, s_i, c_i) \tag{153}$$

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{154}$$

  where s_i is the RNN [decoder] hidden state at time i.

  - NOTE: c_i is *not* the ith element of the standard context vector; rather, it is *itself* a distinct context vector that depends on a sequence of **annotations** $(h_1, \ldots, h_{T_x})$. It seems that each annotation h_i is a hidden (encoder) state "that contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence."
  - The context vector c_i is computed as follows:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{155}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{156}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{157}$$

    where the function e_{ij} is given by an **alignment model** which scores how well the inputs around position j and the output at position i match.

- **Encoder**. It's just a bidirectional RNN. What they call "annotation h_j" is literally just a concatenated vector of $h_j^{forward}$ and $h_j^{backward}$

[48] By "align" the authors are referring to aligning the source-search to the relevant parts for prediction.

(Appendix A). Explained with the TensorFlow user in mind.

Decoder Internals. It's just a GRU. However, it will be helpful to detail how we format the inputs (given we now have attention). Wherever we'd usually pass the previous decoder state s_{i-1}, we now pass a *concatenated* state, $[s_{i-1},\ c_i]$, that also contains the ith context vector. Below I go over the flow of information from GRU input to output:

1. **Notation**: y_t is the loop-embedded output of the decoder (prediction) at time t, s_t is the internal hidden state of the decoder at time t, and c_t is the context vector at time t. $\tilde{s}_t$ is the proposed/proposal state at time t.
2. **Gates**:

$$z_t = \sigma\left(W_z y_{t-1} + U_z[s_{t-1},\ c_t]\right) \qquad \textbf{[update gate]} \tag{158}$$

$$r_t = \sigma\left(W_r y_{t-1} + U_r[s_{t-1},\ c_t]\right) \qquad \textbf{[reset gate]} \tag{159}$$

$$\tag{160}$$

3. **Proposal state**:

$$\tilde{s}_t = \tanh\left(W y_{t-1} + U[r_t \circ s_{t-1},\ c_t]\right) \tag{161}$$

4. **Hidden state**:

$$s_t = (1 - z_t) \circ s_{t-1} + z_t \circ \tilde{s}_t \tag{162}$$

Alignment Model. All equations enumerated below are for some timestep t during the decoding process.

1. **Score**: For all $j \in [0, L_{enc}-1]$ where L_{enc} is the number of words in the encoder sequence, compute:

$$a_j = a(s_{t-1}, h_j) = v_a^T \tanh\left(W_a s_{t-1} + U_a h_j\right) \tag{163}$$

2. **Alignments**: Feed the unnormalized alignments (scores) through a softmax so they represent a valid probability distribution.

$$a_j \leftarrow \frac{e^{a_j}}{\sum_{k=0}^{L_{enc}-1} e^{a_k}} \tag{164}$$

3. **Context**: The context vector input for our decoder at this timestep:

$$c = \sum_{j=1}^{L_{enc}} a_j h_j \tag{165}$$

Decoder Outputs. All below is for some timestep t during the decoding process. To find the probability of some (one-hot) word y [at timestep t]:

$$\Pr(y \mid s, c) \propto e^{y^T W_o u} \tag{166}$$

$$u = [\max\{\tilde{u}_{2j-1}, \tilde{u}_{2j}\}]_{j=1,\ldots,\ell}^{T} \tag{167}$$

$$\tilde{u} = U_o[s_{t-1},\ c] + V_o y_{t-1} \tag{168}$$

N.B.: From reading other (and more recent) papers, these last few equations do not appear to be the way it is usually done (thank the lord). See Luong's work for a much better approach.

Effective Approaches to Attention-Based NMT

[Luong et. al, 2015]

Attention-Based Models. For attention especially, the devil is in the details, so I'm going to go into somewhat excruciating detail here to ensure no ambiguities remain. For both global and local attention, the following information holds true:

- "At each time step t in the decoding phase, both approaches first take as input the hidden state $\boldsymbol{h}_t$ at the top layer of a stacking LSTM."
- Then, they derive [with different methods] a context vector $\boldsymbol{c}_t$ to capture source-side info.
- Given $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$, they both compute the **attentional hidden state** as:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{169}$$

- Finally, the predictive distribution (decoder outputs) is given by feeding this through a softmax:

$$p(y_t \mid y_{<t}, x) = \text{softmax}\left(\boldsymbol{W}_s\tilde{\boldsymbol{h}}_t\right) \tag{170}$$

Global Attention. Now I'll describe in detail the processes involved in $\boldsymbol{h}_t \to \boldsymbol{a}_t \to \boldsymbol{c}_t \to \tilde{\boldsymbol{h}}_t$.

1. $\boldsymbol{h}_t$: Compute the hidden state $\boldsymbol{h}_t$ in the normal way (not obvious if you've read e.g. Bahdanau's work...)
2. $\boldsymbol{a}_t$:
   (a) Compute the **scores** between $\boldsymbol{h}_t$ and each source $\bar{\boldsymbol{h}}_s$, where our options are:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T\bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T\boldsymbol{W}_a\bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T\tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{171}$$

   (b) Compute the **alignment vector** $\boldsymbol{a}_t$ of length L_{enc} (number of words in the encoder sequence):

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{172}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'}\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{173}$$

3. $\boldsymbol{c}_t$: The weighted average over all source (encoder) hidden states[49]:

$$\boldsymbol{c}_t = \sum_{i=1}^{L_{enc}} \boldsymbol{a}_t(i)\bar{\boldsymbol{h}}_i \tag{174}$$

4. $\tilde{\boldsymbol{h}}_t$: For convenience, I'll copy the equation for $\tilde{\boldsymbol{h}}_t$ again here:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{175}$$

Input-Feeding Approach. A copy of each output $\tilde{\boldsymbol{h}}_t$ is sent forward and concatenated with the inputs for the next timestep, i.e. the inputs go from $\boldsymbol{x}_{t+1}$ to $[\tilde{\boldsymbol{h}}_t; \boldsymbol{x}_{t+1}]$.

[49]NOTE: Right after mentioning the context vector, the authors have the following cryptic footnote that may be useful to ponder: *For short sentences, we only use the top part of a_t and for long sentences, we ignore words near the end.*

Using Large Vocabularies for NMT

 Written by Brandon McKinzie

Paper information:

- Full title: On Using Very Large Target Vocabulary for Neural Machine Translation.
- Authors: Jean, Cho, Memisevic, Bengio.
- Date: 18 Mar 2015.
- [arXiv link]

NMT Overview. Typical implementation is encoder-decoder network. Notation for inputs & encoder:

$$x = (x_1, \ldots, x_T) \qquad \textbf{[source sentence]} \tag{176}$$

$$h = (h_1, \ldots, h_T) \qquad \textbf{[encoder state seq]} \tag{177}$$

$$h_t = f(x_t, h_{t-1}) \tag{178}$$

where f is the function defined by the *cell state* (e.g. GRU/LSTM/etc.). Then the decoder generates the output sequence y, and with probability given below:

$$y = (y_1, \ldots, y'_T) \qquad [y_i \in \mathbb{Z}] \tag{179}$$

$$\Pr[y_t \mid y_{<t},\ x] \propto e^{q(y_{t-1},\ z_t,\ c_t)} \tag{180}$$

$$z_t = g(y_{t-1}, z_{t-1}, c_t) \qquad \textbf{[decoder hidden?]} \tag{181}$$

$$c_t = r(z_{t-1}, h_1, \ldots, h_T) \qquad \textbf{[decoder inp?]} \tag{182}$$

The functions q, g, and r are just placeholders – "some function of [inputs]."

As usual, model is jointly trained to maximize the conditional log-likelihood of correct translation. For N training sample pairs (x^n, y^n), and denoting the length of the n-th target sentence as T_n, this can be written as,

$$\theta^* = \arg\max_{\theta} \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log\left(\Pr[y_t^n \mid y_{<t}^n,\ x^n]\right) \tag{183}$$

Model Details. Above is the general structure. Here I'll summarize the specific model chosen by the authors.

- **Encoder**. Bi-directional, which just means $h_t = \left[h_t^{backward};\ h_t^{forward}\right]$. The chosen cell state (the function f) is GRU.
- **Decoder**. At each timestep, computes the following:
  → **Context vector c_t**.

$$c_t = \sum_{i=1}^{T} \alpha_i h_i \tag{184}$$

$$\alpha_t = \frac{e^{a(h_t, z_{t-1})}}{\sum_k e^{a(h_k, z_{t-1})}} \tag{185}$$

a is a standard single-hidden-layer NN.

  → **Decoder hidden state z_t**. Also a GRU cell. Computed based on the previous hidden state z_{t-1}, the previously generated symbol y_{t-1}, and also the computed context vector c_t.
- **Next-word probability**. They model equation 180 as[50] ,

Reminder: y_i is an integer token, while $\mathbf{w}_i$ is the target vector of length vocab size

$$\Pr[y_t \mid y_{<t},\ x] = \frac{1}{Z} e^{\mathbf{w}_t^T \phi(y_{t-1}, z_t, c_t) + b_t} \tag{186}$$

$$Z = \sum_{k:\ y_k \in V} e^{\mathbf{w}_k^T \phi(y_{t-1}, z_t, c_t) + b_k} \tag{187}$$

where ϕ is affine transformation followed by a nonlinear activation, $\mathbf{w}_t$ and b_t are the **target word vector** and bias. V is the set of all target *vocabulary*.

Approximate Learning Approach. Main idea:

> "In order to avoid the growing complexity of computing the normalization constant, we propose here to use only a small subset V' of the target vocabulary at each update."

Consider the gradient of the log-likelihood[51], written in terms of the energy $\mathcal{E}$.

$$\nabla \log\left(\Pr[y_t \mid y_{<t},\ x]\right) = \nabla \mathcal{E}(y_t) - \sum_{k:\ y_k \in V} \Pr[y_k \mid y_{<t},\ x] \nabla \mathcal{E}(y_k) \tag{188}$$

$$\mathcal{E}(y_j) = \mathbf{w}_j^T \phi(y_{t-1}, z_t, c_t) + b_j \tag{189}$$

[50] Note: The formula for Z is correct. Notice that the only part of the RHS of $\Pr(y_t)$ with a t is as the subscript of w. To be clear, w_k is a full word vector and the sum is over all words in the output *vocabulary*, the index k has absolutely nothing to do with timestep. They use the word target but make sure not to misinterpret that as somehow meaning target words in the sentence or something.

[51] **NOTE TO SELF**: After long and careful consideration, I'm concluding that the authors made a typo when defining $\mathcal{E}(y_j)$, which they choose to subscript all parts of the RHS with j, but that is in direct contradiction with a step-by-step derivation, which is why I have written it the way it is. I'm pretty sure my version is right, but I know you'll have to re-derive it yourself next time you see this. And you'll somehow prove me wrong. Actually, after reading on further, I doubt you'll prove me wrong. Challenge accepted, me. Have fun!

The crux of the approach is interpreting the second term as $\mathbb{E}_P[\nabla\mathcal{E}(y)]$, where P denotes $Pr(y \mid y_{<t}, x)$. They approximate this expectation by taking it over a <u>subset</u> V' of the predefined proposal <u>distribution</u> Q. So Q is a p.d.f. over the possible y_i, and we sample *from* Q to generate the elements of the subset V'.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] \approx \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{190}$$

$$\omega_k = e^{\mathcal{E}(y_k) - \log Q(y_k)} \tag{191}$$

Here is some math I did that was illuminating to me; I'm not sure why the authors didn't point out these relationships.

$$\omega_k = \frac{e^{\mathcal{E}(y_k)}}{Q(y_k)} \quad \text{thus} \quad p(y_k \mid y_{<t}, x) = \omega_k \frac{Q(y_k)}{Z} \tag{192}$$

$$\rightarrow \quad e^{\mathcal{E}(y_k)} = Z \cdot p(y_k \mid y_{<t}, x) = Q(y_k) \cdot \omega_k \tag{193}$$

Now *check this out*

Below are the exact and approximate formulas for $\mathbb{E}_P[\nabla\mathcal{E}(y)]$ written in a ~~seductive~~ suggestive manner. Pay careful attention to subscripts and primes.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V} \frac{\omega_k \cdot Q(y_k)}{\sum_{k':y_{k'}\in V} \omega_{k'} \cdot Q(y_{k'})} \nabla\mathcal{E}(y_k) \tag{194}$$

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{195}$$

They're almost the same! It's much easier to see why when written this way. I interpret the difference as follows: in the exact case, we explicitly attach the probabilities $Q(y_k)$ and sum over all values in V. In the second case, by sampling a subset V' from Q, we have encoded these probabilities implicitly as the relative frequency of elements y_k in V'

How to do in practice (very important).

> "In practice, we partition the training corpus and define a subset V' of the target vocabulary for each partition prior to training. Before training begins, we sequentially examine each target sentence in the training corpus and accumulate unique target words until the number of unique target words reaches the predefined threshold τ. The accumulated vocabulary will be used for this partition of the corpus during training. We repeat this until the end of the training set is reached. Let us refer to the subset of target words used for the i-th partition by V'_i.

Candidate Sampling – TensorFlow

 Written by Brandon McKinzie

[Link to article]

What is Candidate Sampling The goal is to learn a compatibility function $F(x, y)$ which says something about the compatibility of a class y with a context x. Candidate sampling: for each training example (x_i, y_i), only need to evaluate $F(x, y)$ for a small set of classes $\{C_i\} \subset \{L\}$, where $\{L\}$ is the set of all possible classes (vocab size number of elements). We represent $F(x, y)$ as a *layer that is trained by back-prop from/within the loss function.*

C.S. for Sampled Softmax. I'll further narrow this down to my use case of having exactly 1 target class (word) at a given time. Any other classes are referred to as **negative** classes (for that example).

Sampling algorithm. For each training example (x_i, y_i), do:

- Sample the subset $S_i \subset L$. How? By sampling from $Q(y|x)$ which gives the probability of any particular y being included in S_i.
- Create the set of **candidates**, which is just $C_i := S_i \cup y_i$.

Training task. We are given this set C_i and want to find out which element of C_i is the target class y_i. In other words, we want the posterior probability that any of the y in C_i are the target class, given what we know about C_i and x_i. We can evaluate and rearrange as usual with Bayes' rule to get:

$$\Pr\left(y_i^{true} = y \mid C_i,\ x_i\right) = \frac{\Pr\left(y_i^{true} = y \mid x_i\right) \cdot \Pr\left(C_i \mid y_i^{true} = y,\ x_i\right)}{\Pr\left(C_i \mid x_i\right)} \tag{196}$$

$$= \frac{\Pr\left(y \mid x_i\right)}{Q\left(y \mid x_i\right)} \cdot \frac{1}{K(x_i, C_i)} \tag{197}$$

where they've just defined

$$K(x_i, C_i) \triangleq \frac{\Pr\left(C_i \mid x_i\right)}{\prod_{y' \in C_i} Q(y' \mid x_i) \prod_{y' \in (L - C_i)} \left(1 - Q(y' \mid x_i)\right)} \tag{198}$$

Clarifications.

- The learning function $F(x, y)$ is the *input* to our softmax. <u>It is our neural network</u>, excluding the softmax function.
- After training our network, it should have learned the general form

$$F(x, y) = \log(\Pr(y \mid x)) + K(x) \tag{199}$$

which is the general form because

$$\text{Softmax}(\log(\Pr(y \mid x)) + K(x)) = \frac{e^{\log(\Pr(y|x))+K(x)}}{\sum_{y'} e^{\log(\Pr(y'|x)+K(x)}} \tag{200}$$

$$= \Pr(y \mid x) \tag{201}$$

Note that I've been a little sloppy here, since $\Pr(y \mid x)$ up until the last line actually represented the (possibly) unnormalized/*relative* probabilities.

- **[MAIN TAKEAWAY]**. Time to bring it all together. Notice that we've only trained $F(x, y)$ to include *part* of what's needed to compute the probability of any y being the target given x_i <u>and C_i</u> ...equation 199 doesn't take into account C_i at all! Luckily we know the form of the full equation because it just the log of equation 197. We can easily satisfy that by subtracting $\log(Q(y \mid x))$ from $F(x, y)$ right before feeding into the softmax.

TL;DR. Train network to learn $F(x, y)$ before softmax, but instead of feeding $F(x, y)$ to softmax directly, feed

$$\text{Softmax Input: } F(x, y) - \log(Q(y \mid x)) \tag{202}$$

instead. That's it.

Attention Terminology

Written by Brandon McKinzie

Generally useful info. Seems like there are a few notations floating around, and here I'll attempt to set the record straight. The order of notes here will loosely correspond with the order that they're encountered going from encoder output to decoder output.

Jargon. The people in the attention business *love* obscure names for things that don't need names at all. Terminology:

- **Attentions keys/values**: Encoder output sequence.
- **Query**: Decoder [cell] state. Typically the most recent one.
- **Scores**: Values of e_{ij}. For the Bahdanau version, in code this would be computed via

$$e_i = v^T \tanh(\mathrm{FC}(s_{i-1}) + \mathrm{FC}(h)) \tag{203}$$

  where we'd have FC be tf.layers.fully_connected with num_outputs equal to our attention size (up to us). Note that v is a vector.
- **Alignments**: output of the softmax layer on the attention scores.
- **Memory**: The α matrix in the equation $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$.

When someone lazily calls some layer output the "attention", they are usually referring to the layer *just after* the linear combination/map of encoder hidden states. You'll often see this as some vague function of the previous decoder state, context vector, and possibly even decoder output (after project), like $f(s_{i-1}, y_{i-1}, c_i)$. In 99.9% of cases, this function is just a fully connected layer (if even needed) to map back to the state size for decoder input. That is it.

From encoder to decoder. The path of information flow from encoder outputs to decoder inputs, a non-trivial process that isn't given the *attention* (heh) it deserves[52]

1. **Encoder outputs**. Tensor of shape **[batch size, sequence length, state size]**. The state is typically some RNNCell state.
   - Note: TensorFlow's AttenntionMechanism classes will actually convert this to **[batch size, L_{enc}, attention size]**, and refer to it as the "memory". It is also what is returned when calling myAttentionMech.values.

[52] For some reason, the literature favors explaining the path "backwards", starting with the highly abstracted "decoder inputs as a weighted sum of encoder states" and then breaking down what the weights are. Unfortunately, the weights are computed via a multi-stage process so that becomes very confusing very quick.

2. **Compute the scores**. The attention scores are the computation described by Luong/Bahdanau techniques. They both take an inner product of sorts on *copies* of the encoder outputs and decoder previous state (query). The main choices are:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{204}$$

Synonyms:
- scores
- unnormalized alignments

where the shapes are as follows (for single timestep during decoding process):

- $\bar{\boldsymbol{h}}_s$: **[batch size, 1, state size]**
- $\boldsymbol{h}_t$: **[batch size, 1, state size]**
- $\boldsymbol{W}_a$: **[batch size, state size, state size]**
- $\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)$: **[batch size]**

Synonyms:
- softmax outputs
- attention dist.
- alignments

3. **Softmax the scores**. In the vast majority of cases, the attention scores are next fed through a softmax to convert them into a valid probability distribution. Most papers will call this some vague probability function, when in reality they are using softmaxonly.

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{205}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{206}$$

where the alignment vector $\boldsymbol{a}_t$ has shape **[batch size,** L_{enc}**]**

Synonyms:
- context vector
- attention

4. **Compute the context vector**. The inner product of the softmax outputs and the raw encoder outputs. This will have shape **[batch size, attention size]** in TensorFlow, where attention size is from the constructor for your AttentionMechanism.

5. **Combine context vector and decoder output**: Typically with a concat. *The result is what people mean when they say "attention"*. Luong et al. denotes this as $\tilde{\boldsymbol{h}}_t$, the decoder output at timestep t. This is what TensorFlow means by "Luong-style mechanisms output the attention." And yes, these are used (at least for Luong) to compute the prediction:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c\left[\boldsymbol{c}_t,\ \boldsymbol{h}_t\right]\right) \tag{207}$$

$$p(y_t \mid y_{<t}, x) = \text{softmax}(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t) \tag{208}$$

TextRank

Introduction. A graph-based ranking algorithm is a way of deciding on the importance of a vertex within a graph, by taking into account global information recursively computed from the entire graph, rather than relying only on local vertex-specific information. **TextRank** is a graph-based ranking model for graphs extracted from natural language texts. The authors investigate/evaluate TextRank on unsupervised keyword and sentence extraction.

Semantic graph: one whose structure encodes meaning between the nodes (semantic elements).

The ~~TextRank~~ PageRank Model. In general [graph-based ranking], a vertex can be ranked based on certain properties such as the number of vertices pointing to it (in-degree), how highly-ranked *those* vertices are, etc. Formally, the authors [of *PageRank*] define the score of a vertex V_i as follows:

$$S(V_i) = (1-d) + d * \sum_{V_j \in \text{In}(V_i)} \frac{1}{\mid \text{Out}(V_i) \mid} S(V_j) \quad \text{where} \quad d \in \Re[0,1] \tag{209}$$

The factor d is usually set to 0.85.

and the damping factor d is interpreted as the probability of jumping from a given vertex[53] to another <u>random</u> vertex in the graph. In practice, the algorithm is implemented through the following steps:

(1) Initialize all vertices with arbitrary values.[54]
(2) Iterate over vertices, computing equation 4.9 until convergence [of the error rate] below a predefined threshold. The error-rate, defined as the difference between the "true score" and the score computed at iteration k, $S^k(V_i)$, is *approximated* as:

$$\text{Error}^k(V_i) \approx S^k(V_i) - S^{k-1}(V_i) \tag{210}$$

[53] Note that d is a single parameter for the graph, i.e. it is the same for all vertices.

[54] The authors do not specify what they mean by arbitrary. What range? What sampling distribution? Arbitrary as in uniformly random? **EDIT**: The authors claim that the vertex values upon completion are not affected by the choice of initial value. Investigate!

Weighted Graphs. In contrast with the PageRank model, here we are concerned with natural language texts, which may include multiple or partial links between the units (vertices). The authors hypothesize that modifying equation 4.9 to incorporate *weighted* connections may be useful for NLP applications.

$$WS(V_i) = (1-d) + d * \sum_{j \in \text{In}(V_i)} \frac{w_{ji}}{\sum V_k \in \text{Out}(V_j) w_{jk}} WS(V_j) \tag{211}$$

w_{ij} denotes the connection between vertices V_i and V_j.

where I've shown the modified part in green. The authors mention they set all weights to random values in the interval 0-10 (no explanation).

Text as a Graph. In general, the application of graph-based ranking algorithms to natural language texts consists of the following main steps:

(1) Identify text units that best define the task at hand, and add them as vertices in the graph.
(2) Identify relations that connect such text unit in order to draw edges between vertices in the graph. Edges can be directed or undirected, weighted or unweighted.
(3) Iterate the algorithm until convergence.
(4) Sort [in reversed order] vertices based on final score. Use the values attached to each vertex for ranking/selection decisions.

Graph. The authors apply TextRank to extract words/phrases that are representative for a given document. The individual graph components are defined as follows:

- **Vertex**: sequence of one or more lexical units from the text.
  - In addition, we can restrict which vertices are added to the graph with syntactic filters.
  - Best filter [for the authors]: *nouns and adjectives only.*
- **Edge**: two vertices are connected if their corresponding lexical units co-occur within a window of N words[55].

Typically $N \in \mathbb{Z}[2, 10]$

Procedure:

(1) **Pre-Processing**: Tokenize and annotate [with POS] the text.

(2) **Build the Graph**: Add all [single] words to the graph that pass the syntactic filter, and connect [undirected/unweighted] edges as defined earlier (co-occurrence).

(3) **Run algorithm**: Initialize all scores to 1. For a convergence threshold of 0.0001, usually takes about 20-30 iterations.

(4) **Post-Processing**:
   (i) Keep the top T vertices (by score), where the authors chose $T = |V|/3$.[56] Remember that vertices are still individual words.
   (ii) From the new subset of T keywords, collapse any that were adjacent in the original text in a single lexical unit.

Evaluation. The data set used is a collection of 500 abstracts, each with a set of keywords. Results are evaluated using **precision**, **recall**, and **F-measure**[57]. The best results were obtained with a co-occurrence window of 2 [on an undirected graph], which yielded:

Precision: 31.2% Recall: 43.1% F-measure: 36.2

The authors found that larger window size corresponded with lower precision, and that directed graphs performed worse than undirected graphs.

[55] That is ... simpler than expected. *Can we do better?*

[56] Another approach is to have T be a fixed value, where typically $5 < T < 20$.

[57] Brief terminology review:

- **Precision**: fraction of keywords extracted that are in the "true" set of keywords.
- **Recall**: fraction of "true" keywords that are in the extracted set of keywords.
- **F-score**: combining precision and recall to get a single number for evaluation:

$$F = \frac{2pr}{p + r}$$

A PR Curve plots precision as a function of recall.

Sentence Extraction

Graph. Now we move to "sentence extraction for automatic summarization."

- **Vertex**: a vertex is added to the graph for each sentence in the text.
- **Edge**: each weighted edge represents the similarity between two sentences. The authors use the following similarity measure between two sentences S_i and S_j:

$$\text{Similarity}(S_i, S_j) = \frac{\mid S_i \cap S_j \mid}{\log(\mid S_i \mid) + \log(\mid S_j \mid)} \tag{212}$$

  where the numerator is the number of words that occur in both S_i and S_j.

The **procedure** is identical to the algorithm described for keyword extraction, except we run it on full sentences.

Evaluation. The data set used is 567 news articles. For each article, TextRank generates a 100-word summary (i.e. they set $T = 100$). They evaluate with the Rouge evaluation toolkit (Ngram statistics).

Simple Baseline for Sentence Embeddings

Table of Contents Local

Written by Brandon McKinzie

Overview. It turns out that simply taking a weighted average of word vectors and doing some PCA/SVD is a competitive way of getting unsupervised word embeddings. Apparently it *beats* supervised learning with LSTMs (?!). The authors claim the theoretical explanation for this method lies in a latent variable generative model for sentences (of course).

Discussion based on paper by Arora et al., (2017).

Algorithm.

1. Compute the weighted average of the word vectors in the sentence:

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{213}$$

The authors call their weighted average the **Smooth Inverse Frequency (SIF)**.

   where $\boldsymbol{w}_i$ is the word vector for the ith word in the sentence, a is a parameter, and $p(\boldsymbol{w}_i)$ is the (estimated) word frequency [over the entire corpus].
2. Remove the projections of the average vectors on their first principal component ("common component removal") (y tho?).

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

```
for all sentence s in S do
    v_s ← (1/|s|) Σ_{w∈s} a/(a+p(w)) v_w
end for
Compute the first principal component u of {v_s : s ∈ S}
for all sentence s in S do
    v_s ← v_s − u u^⊤ v_s
end for
```

Theory. Latent variable generative model. The model treats corpus generation as a dynamic process, where the t-th word is produced at time step t, driven by the random walk of a **discourse vector** $c_t \in \Re^d$ (d is size of the embedding dimension). The discourse vector is *not* pointing to a specific word; rather, it describes what is being talked about. We can tell how related (correlation) the discourse is to any word w and corresponding vector v_w by taking the inner product $c_t \cdot v_w$. Similarly, we model the probability of observing word w at time t, w_t, as:

$$\Pr\left[w_t \mid c_t\right] \propto e^{c_t \cdot v_w} \tag{214}$$

- **The Random Walk**. If we assume that c_t doesn't change much over the words in a single sentence, we can assume it stays at some c_s. The authors claim that in their previous paper they showed that the MAP[58] estimate of c_s is – up to multiplication by a scalar – the average of the embeddings of the words in the sentence.
- **Improvements/Modifications to 214**.
  1. Additive term $\alpha p(w)$ where α is a scalar. Allows words to occur even if $c_t \cdot v_w$ is very small.
  2. Common discourse vector $c_0 \in \Re^d$. Correction term for the most frequent discourse that is often related to syntax.
- **Model**. Given the discourse vector c_s for a sentence s, the probability that w is in the sentence (at all (?)):

$$\Pr\left[w \mid c_s\right] = \alpha p(w) + (1-\alpha)\frac{e^{\tilde{c}_s \cdot v_w}}{Z_{\tilde{c}_s}} \tag{215}$$

$$\tilde{c}_s = \beta c_0 + (1-\beta) c_s \tag{216}$$

  with $c_0 \perp c_s$ and $Z_{\tilde{c}_s}$ is a normalization constant, taken over all $w \in V$.

Embedding Calculations. Here I'll go into more depth on how the algorithm described earlier is derived. **TODO**

[58] Review of MAP:

$$\theta_{MAP} = \arg\max_{\theta} \sum_i \log\left(p_X(x \mid \theta) p(\theta)\right)$$

Survey of Text Clustering Algorithms

Aggarwal et al., "A Survey of Text Clustering Algorithms," (2012).

Introduction. The unique characteristics for clustering *text*, as opposed to more traditional (numeric) clustering, are (1) large dimensionality but highly sparse data, (2) words are typically highly correlated, meaning the number of principal components is much smaller than the feature space, and (3) the number of words per document can vary, so we must normalize appropriately.

Common types of clustering algorithms include agglomerative clustering algorithms, partitioning algorithms, and standard parametric modeling based methods such as the EM-algorithm.

Feature Selection and Transformation Methods

Feature Selection.

- **Document Frequency-Based**. Using document frequency to filter *out* irrelevant features. Dealing with certain words, like "the", should probably be taken a step further and simply removed (stop words).
- **Term Strength**. A more aggressive technique for stop-word removal. It's used to measure how informative a word/term t is for identifying two related documents, x and y. Denoted $s(t)$, it is defined as:

  See ref 94 of the paper for more.

  $$s(t) = \Pr\left[t \in y \mid t \in x\right] \tag{217}$$

  So, how do we know x and y are related to begin with? One way is a user-defined cosine similarity threshold. Say we gather a set of such *pairs* and randomly identify one of the pair as the "first" document of the pair, then we can approximate $s(t)$ as

  $$s(t) = \frac{\text{Num pairs in which t occurs in both}}{\text{Num pairs in which t occurs in the first of the pair}} \tag{218}$$

  > *In order to prune features, the term strength may be compared to the expected strength of a term which is randomly distributed in the training documents with the same frequency. If the term strength of t is not at least two standard deviations greater than that of the random word, then it is removed from the collection.*

- **Entropy-Based Ranking**. The quality of a term is measured by the entropy reduction when it is removed [from the collection]. The entropy $E(t)$ of term t in a collection of n

documents is:

$$E(t) = -\sum_{i=1}^{n}\sum_{j=1}^{n}\left(S_{ij}\cdot\log(S_{ij}) + (1-S_{ij})\cdot\log(1-S_{ij})\right) \tag{219}$$

$$S_{ij} = 2^{-d_{ij}/\bar{d}} \tag{220}$$

where

- $S_{ij} \in (0,1)$ is the similarity between doc i and j.
- d_{ij} is the distance between i and j after the term t is removed
- $\bar{d}$ is the average distance between the documents after the term t is removed.

LSI-based Methods. Latent Semantic Indexing is based on dimensionality reduction where the new (transformed) features are a linear combination of the originals. This helps magnify the semantic effects in the underlying data. LSI is quite similar to PCA[59], except that we use an approximation of the covariance matrix C which is appropriate for the sparse and high-dimensional nature of text data.

See ref 28. of paper for more on LSI.

Let $\boldsymbol{A} \in \mathbb{R}^{n\times d}$ be term-document matrix, where $\boldsymbol{A}_{i,j}$ is the (normalized) frequency for term j in document i. Then $\boldsymbol{A}^T\boldsymbol{A} = n\cdot\Sigma$ is the (scaled) approximation to covariance matrix[60], assuming the data is mean-centered. Quick check/reminder:

$$(\boldsymbol{A}^T\boldsymbol{A})_{ij} = \boldsymbol{A}_{:,i}^T\boldsymbol{A}_{:,j} \triangleq \boldsymbol{a}_i^T\boldsymbol{a}_j \tag{221}$$

$$\approx n\cdot\mathbb{E}\left[\mathbf{a}_i\mathbf{a}_j\right] \tag{222}$$

where the expectation is technically over the underlying data distribution, which gives e.g. $P(a_i = x)$, the probability the ith word in our vocabulary having frequency x. Apparently, since the data is sparse, we don't have to worry much about it actually being mean-centered (**why?**).

As usual, we using the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ with the largest variance in order to represent the text[61]. In addition:

> *One excellent characteristic of LSI is that the truncation of the dimensions removes the noise effects of synonymy and polysemy, and the similarity computations are more closely affected by the semantic concepts in the data.*

[59] The difference between LSI and PCA is that PCA subtracts out the means, which destroys the sparseness of the design matrix.

[60] Approximation because it is based on our training data, not on true expectations over the underlying data-distribution.

[61] In typical collections, only about 300 to 400 eigenvectors are required for the representation.

Non-negative Matrix Factorization. Another latent-space method (like LSI), but particularly suitable for clustering. The main characteristics of the NMF scheme:

- In LSI, the new basis system consists of a set of orthonormal vectors. This is *not* the case for NMF.
- In NMF, the vectors in the basis system **directly correspond to cluster topics.** Therefore, the cluster membership for a document may be determined by examining the largest component of the document along any of the [basis] vectors.

Assume we want to create k clusters, using our n documents and vocabulary size d. The goal of NMF is to find matrices $\boldsymbol{U} \in \mathbb{R}^{n\times k}$ and $\boldsymbol{V} \in \mathbb{R}^{d\times k}$ that minimize:

$$J = \frac{1}{2}||\boldsymbol{A} - \boldsymbol{U}\boldsymbol{V}^T||_F^2 \tag{223}$$

$$= \frac{1}{2}\Big(tr(\boldsymbol{A}\boldsymbol{A}^T) - 2tr(\boldsymbol{A}\boldsymbol{V}\boldsymbol{U}^T) + tr(\boldsymbol{U}\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{U}^T)\Big) \tag{224}$$

$u_{ij} \geq 0$

$v_{ij} \geq 0$

Note that the columns of $\boldsymbol{V}$ provide the k basis vectors which correspond to the k different clusters. We can interpret this as trying to factorize $\boldsymbol{A} \approx \boldsymbol{U}\boldsymbol{V}^T$. For each row, $\boldsymbol{a}$, of $\boldsymbol{A}$ (document vector), this is

$$\boldsymbol{a} \approx \boldsymbol{u} \cdot \boldsymbol{V}^T \tag{225}$$

$$= \sum_{i=1}^{k} \boldsymbol{u}_i \boldsymbol{V}_i^T \tag{226}$$

Therefore, the document vector $\boldsymbol{a}$ can be rewritten as an approximate linear (non-negative) combination of the basis vector which corresponds to the k columns of $\boldsymbol{V}^T$.

Langrange-multiplier stuff: Our optimization problem can be solved using the Lagrange method.

- Variables to optimize: All elements of both $\boldsymbol{U} = [u_{ij}]$ and $\boldsymbol{V} = [v_{ij}]$
- Constraint: non-negativity, i.e. $\forall i, j,\ u_{ij} \geq 0$ and $v_{ij} \geq 0$.
- Multipliers: Denote as <u>matrices</u> α and β, with same dimensions as $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively.
- Lagrangian: I'll just show it here first, and then explain in this footnote[62]:

$$\mathcal{L} = J + tr(\alpha \cdot \boldsymbol{U}^T) + tr(\beta \cdot \boldsymbol{V}^T) \tag{227}$$

$$\text{where} \quad tr(\alpha \cdot \boldsymbol{U}^T) = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{u}_i = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_{ij} u_{ij} \tag{228}$$

Any matrix multiplication with a $\cdot$ is just a reminder to think of the matrices as column vectors.

You should think of α as a column vector of length n, and $\boldsymbol{U}^T$ as a row vector of length n. The reason we prefer $\mathcal{L}$ over just J is because now we have an *unconstrained* optimization problem.

[62] Recall that in Lagrangian minimization, $\mathcal{L}$ takes the form of [the-function-to-be-minimized] + λ ([constraint-function] - [expected-value-of-constraint-at-optimum]). So the second term is expected to tend toward zero (i.e. critical point) at the optimal values. In our case, since our optimal value is sort-of (**?**) at 0 for any value of u_{ij} and/or v_{ij}, we just have a sum over [langrange-mult] $\times$ [variable].

- Optimization: Set the partials of $\mathcal{L}$ w.r.t both U and V (separately) to zero[63]:

$$\frac{\partial \mathcal{L}}{\partial U} = -\boldsymbol{A} \cdot \boldsymbol{V} + \boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V} + \boldsymbol{\alpha} = 0 \tag{229}$$

$$\frac{\partial \mathcal{L}}{\partial V} = -\boldsymbol{A}^T \cdot \boldsymbol{U} + \boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U} + \boldsymbol{\beta} = 0 \tag{230}$$

  Since, ultimately, these just say [some matrix] = 0, we can multiply both sides (element-wise) by a constant ($x \times 0 = 0$). *Using*[64] the **Kuhn-Tucker conditions** $\alpha_{ij} \cdot u_{ij} = 0$ and $\beta_{ij} \cdot v_{ij} = 0$, we get:

$$(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij} - (\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij} \cdot u_{ij} = 0 \tag{231}$$

$$(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} - (\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} = 0 \tag{232}$$

- Update rules:

$$u_{ij} = \frac{(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij}}{(\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij}} \tag{233}$$

$$v_{ij} = \frac{(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij}}{(\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij}} \tag{234}$$

Distance-based Clustering Algorithms

One challenge in clustering short segments of text (e.g., tweets) is that exact keyword matching may not work well. One general strategy for solving this problem is to expand text representation by exploiting related text documents, which is related to smoothing of a document language model in information retrieval.

See ref. 66 in the paper for computing similarities of short text segments.

Agglomerative and Hierarchical Clustering. "Agglomerative" refers to the process of bottom-up clustering to build a tree – at the bottom are leaves (documents) and internal nodes correspond to the merged groups of clusters. The different methods for merging groups of documents for the different agglomerative methods are as follows:

- **Single Linkage Clustering**. Defines similarity between two groups (clusters) of documents as the largest similarity between any pair of documents from these two groups. First, (1) compute all similarity pairs [between documents; ignore cluster labels], then (2) sort in decreasing order, and (3) walk through the list in that order, merging clusters if the pair belong to different clusters. One drawback is *chaining*: the resulting clusters assume transitivity of similarity[65].

[63] Recall that the Lagrangian consists entirely of traces (re: scalars). Derivatives of traces with respect to matrices output the same dimension as that matrix, and derivatives are taken element-wise as always.

[64] i.e. the equations that follow are *not* the KT conditions, they just *use*/exploit them...

[65] Here, transitivity of similarity means if A is similar to B, and B is similar to C, then A is similar to C. This is not guaranteed by any means for textual similarity, and so we can end up with A and Z in the same cluster, even though they aren't similar at all.

- **Group-Average Linkage Clustering**. Similarity between two clusters is the *average* similarity over all unique pairwise combinations of documents from one cluster to the other. One way to speed up this computation with an approximation is to just compute the similarity between the mean vector of either cluster.
- **Complete Linkage Clustering**. Similarity between two clusters is the *worst-case* similarity between any pair of documents.

Distance-Based Partitioning Algorithms.

- **K-Medoid Clustering**. Use a set of points from training data as anchors (medoids) around which the clusters are built. Key idea is we are using an optimal set of representative documents *from the original corpus*. The set of k reps is successively improved via randomized inter-changes. In each iteration, we replace a randomly picked rep in the current set of medoids with a randomly picked rep from the collection, if it improves the clustering objective function. This approach is applied until convergence is achieved.

K-Medoid isn't great for clustering text, especially short texts.

- **K-Means Clustering**. Successively (1) assigning points to the nearest cluster centroid and then (2) re-computing the centroid of each cluster. Repeat until convergence. Requires typically few iterations (about 5 for many large data sets). Disadvantage: sensitive to initial set of seeds (initial cluster centroids). One method for improving the initial set of seeds is to use some supervision - e.g. initialize with k pre-defined topic vectors (see ref. 4 in paper for more).

Hybrid Approach: Scatter-Gather Method. Use a hierarchical clustering algorithm on a sample of the corpus in order to find a robust initial set of seeds. This robust set of seeds is used in conjunction with a standard k-means clustering algorithm in order to determine good clusters.**TODO:** resume note-taking; page 19/52 of PDF.

Scatter-Gather is discussed in detail in ref. 25 of the paper

---

Probabilistic Document Clustering and Topic Models

Overview. Primary assumptions in any topic modeling approach:

From pg. 31/52 of paper.

- The n documents in the corpus are assumed to each have a probability of belonging to one of k topics. Denote the probability of document D_i belonging to topic T_j as $\Pr[T_j \mid D_i]$. This allows for *soft cluster membership* in terms of probabilities.
- Each topic is associated with a probability vector, which quantifies the probability of the different terms in the lexicon for that topic. For example, consider a document that belongs completely to topic T_j. We denote the probability of term t_l occurring in that document as $\Pr[t_l \mid T_j]$.

The two main methods for topic modeling are **Probabilistic Latent Semantic Indexing** (PLSA) and **Latent Dirichlet Allocation** (LDA).

PLSA. We note that the two aforementioned probabilities, $\Pr[T_j \mid D_i]$ and $\Pr[t_l \mid T_j]$ allow us to calculate $\Pr[t_l \mid D_i]$: the probability that term t_l occurs in some document D_i:

$$\Pr[t_l \mid D_i] = \sum_{j=1}^{k} \Pr[t_l \mid T_j] \cdot \Pr[T_j \mid D_i] \tag{235}$$

which should be interpreted as a weighted average[66]. From here, we can generate a $n \times d$ matrix of probabilities.

Recall that we also have our $n \times d$ term-document matrix $\boldsymbol{X}$, where $\boldsymbol{X}_{i,l}$ gives the number of times term l occurred in document D_i. This allows us to do maximum likelihood! Our negative log-likelihood, J can be derived as follows:

Interpret $\Pr[\boldsymbol{X}]$ as the joint probability of observing the words in our data and with their assoc. frequencies.

$$\begin{aligned} J &= -\log(\Pr[\boldsymbol{X}]) && (236) \\ &= -\log\left(\prod_{i,l} \Pr[t_l \mid D_i]^{\boldsymbol{X}_{i,l}}\right) && (237) \\ &= -\sum_{i,l} \boldsymbol{X}_{i,l} \cdot \log(\Pr[t_l \mid D_i]) && (238) \end{aligned}$$

and we can plug-in eqn 235 to for evaluating $\Pr[t_l \mid D_i]$. We want to optimize the value of J, subject to the constraints:

$$(\forall T_j): \sum_{l} \Pr[t_l \mid T_j] = 1 \qquad (\forall D_i): \sum_{j} \Pr[T_j \mid D_i] = 1 \tag{239}$$

This can be solved with a Lagrangian method, similar to the process for NMF described earlier. See page 33/52 of the paper for details.

Latent Dirichlet Allocation (LDA). The term-topic probabilities and topic-document probabilities are modeled with a *Dirichlet distribution* as a prior[67]. Typically preferred over PLSI because PLSI more prone to overfitting.

[66] This is actually pretty bad notation, and borderline incorrect. $\Pr[T_j \mid D_i]$ is NOT a conditional probability! It is our prior! It is literally $\Pr[\text{ClusterOf}(D_i) = T_j]$.

[67] LDA is the Bayesian version of PLSI

Online Clustering with Text Streams

Reference List: [3]: A Framework for Clustering Massive Text and Categorical Data Streams; [112]: Efficient Streaming Text Clustering; [48]: Bursty feature representation for clustering text streams; [61]: Clustering Text Data Streams (Liu et al.)

Overview. Maintaining text clusters in real time. One method is the **Online Spherical K-Means Algorithm** (OSKM)[68]. See ref. 112 for more on OSKM

Condensed Droplets Algorithm. I'm calling it that because they don't call it anything – it is the algorithm in [3].

- **Fading function**: $f(t) = 2^{-\lambda \cdot t}$. A time-dependent weight for each data point (text stream). Non-monotonic decreasing; decays uniformly with time.
- **Decay rate**: $\lambda = 1/t_0$. Inverse of the half-life of the data stream.

When a cluster is created by a new point, it is allowed to remain as a trend-setting outlier for at least one half-life. During that period, if at least one more data point arrives, then the cluster becomes an active and mature cluster. If not, the trend-setting outlier is recognized as a true anomaly and is removed from the list of current clusters (*cluster death*). Specifically, this happens when the (weighted) number of points in the [single-point] cluster is 0.5. The same criterion is used to define the death of mature clusters. The statistics of the data points are referred to as **condensed droplets**, which represent the word distributions within a cluster, and can be used in order to compute the similarity of an incoming data point to the cluster. Main idea of algorithm is as follows:

1. Initialize empty set of clusters $\mathcal{C} = \{\}$. As new data points arrive, unit clusters containing individual data points are created. Once a maximum number k of such clusters have been created, we can begin the process of online cluster maintenance.
2. For a new data point X, compute its similarity to each cluster C_j, denoted as $S(X, C_j)$.
   - If $S(X, C_{best}) > \text{thresh}_{outlier}$, or if there are no inactive clusters left[69], insert X to the cluster with maximum similarity.
   - Otherwise, a new cluster is created[70] containing the solitary data point X.

[68] Authors only provide a very brief description, which I'll just copy here:
This technique divides up the incoming stream into small segments, each of which can be processed effectively in main memory. A set of k-means iterations are applied to each such data segment in order to cluster them. The advantage of using a segment-wise approach for clustering is that since each segment can be held in main memory, we can process each data point multiple times as long as it is held in main memory. In addition, the centroids from the previous segment are used in the next iteration for clustering purposes. A decay factor is introduced in order to age- out the old documents, so that the new documents are considered more important from a clustering perspective.

[69] We specify some max allowed number of clusters k.

[70] The new cluster replaces the least recently updated inactive cluster.

Misc.

- Mandatory read: reference [61]. Details phrase extraction/**topic signatures**. The use of using phrases instead of individual words is referred to as **semantic smoothing**.
- For *dynamic* (and more recent) topic modeling, see reference [107] of the paper, titled "A probabilistic model for online document clustering with application to novelty detection."

Semi-Supervised Clustering

Overview. Useful when we have any prior knowledge about the kinds of clusters available in the underlying data. Some approaches:

- Incorporate this knowledge when seeding the cluster centroids for k-means clustering.
- Iterative EM approach: unlabeled documents are assigned labels using a naive Bayes approach on the currently labeled documents. These newly labeled documents are then again used for re-training a Bayes classifier. Iterate to convergence.
- Graph-based approach: graph nodes are documents and the edges are similarities between the connected documents (nodes). We can incorporate prior knowledge by adding certain edges between nodes that we know are similar. A *normalized cut algorithm* is then applied to this graph in order to create the final clustering.

We can also use partially supervised methods in conjunction with pre-existing categorical *hierarchies*.

Deep Sentence Embedding Using LSTMs

 Written by Brandon McKinzie

Palangi et al., "Deep Sentence Embeddings Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval," (2016).

Abstract. Sentence embeddings using LSTM cells, which automatically attenuate unimportant words and detect salient keywords. Main emphasis on applications for document retrieval (matching a query to a document[71]).

Introduction. Sentence embeddings are learned using a loss function defined on *sentence pairs*. For example, the well-known Paragraph Vector[72] is learned in an unsupervised manner as a distributed representation of sentences and documents, which are then used for sentiment analysis.

The authors appear to use a dataset of their own containing examples of (search-query, clicked-title) for a search engine. Their training objective is to maximize the similarity between the two vectors mapped by the LSTM-RNN from the query and the clicked document, respectively. One very interesting claim to pay close attention to:

> *We further show that different cells in the learned model indeed correspond to* ***different topics****, and the keywords associated with a similar topic activate the same cell unit in the model.*

Related Work. (Identified by reference number)

- [2] Good for sentiment, but doesn't capture fine-grained sentence structure.
- [6] Unsupervised embedding method trained on the BookCorpus [7]. Not good for document retrieval task.
- [9] Semi-supervised Recursive Autoencoder (RAE) for sentiment prediction.
- [3] DSSM (uses bag-of-words) and [10] CLSM (uses bag of n-grams) models for IR and also sentence embeddings.
- [12] Dynamic CNN for sentence embeddings. Good for sentiment prediction and question type classification. In [13], a CNN is proposed for **sentence matching**[73]

[71] Note that this similar to topic extraction.
[72] Q. V. Le and T. Mikolov, Distributed representations of sentences and documents.
[73] Might want to look into this.

Basic RNN. The information flow (sequence of operations) is enumerated below.

1. Encode tth word [of the given sentence] in one-hot vector $\boldsymbol{x}(t)$.
2. Convert $\boldsymbol{x}(t)$ to a <u>letter</u> tri-gram vector $\boldsymbol{l}(t)$ using fixed hashing operator[74] $\boldsymbol{H}$:

$$\boldsymbol{l}(t) = \boldsymbol{H}\boldsymbol{x}(t) \tag{240}$$

3. Compute the hidden state $\boldsymbol{h}(t)$, which is the sentence embedding for $t = T$, the length of the sentence.

$$\boldsymbol{h}(t) = \tanh\left(\boldsymbol{U}\boldsymbol{l}(t) + \boldsymbol{W}\boldsymbol{h}(t-1) + \boldsymbol{b}\right) \tag{241}$$

   where $\boldsymbol{U}$ and $\boldsymbol{W}$ are the usual parameter matrices for the input/recurrent paths, respectively.

LSTM. With peephole connections that expose the internal cell state s to the sigmoid computations. I'll rewrite the standard LSTM equations from my textbook notes, but with the modifications for peephole connections:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} + \sum_j P_{i,j}^f s_j^{(t-1)}\right) \tag{242}$$

$$s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + g_i^{(t)} \odot \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \tag{243}$$

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} + \sum_j P_{i,j}^g s_j^{(t-1)}\right) \tag{244}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} + \sum_j P_{i,j}^o s_j^{(t)}\right) \tag{245}$$

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{246}$$

[74] Details aside, the hashing operator serves to lower the dimensionality of the inputs a bit. In particular we use it to convert one-hot word vectors into their letter tri-grams. For example, the word "good" gets surrounded by hashes, '#good#', and then hashed from the one-hot vector to vectorized tri-grams, "#go", "goo", "ood", "od#".

Learning method. We want to maximize the likelihood of the clicked document given query, which can be formulated as the following optimization problem:

$$L(\mathbf{\Lambda}) = \min_{\mathbf{\Lambda}} \left\{ -\log \prod_{r=1}^{N} \Pr\left[D_r^+ \mid Q_r \right] \right\} = \min_{\mathbf{\Lambda}} \sum_{r=1}^{N} l_r(\mathbf{\Lambda}) \tag{247}$$

$$l_r(\mathbf{\Lambda}) = \log \left(1 + \sum_{j=1}^{n} e^{-\gamma \cdot \Delta_{r,j}} \right) \tag{248}$$

where

- N is the number of (query, clicked-doc) pairs in the corpus, while n is the number of negative samples used during training.
- D_r^+ is the clicked document for rth query.
- $\Delta_{r,j} = R(Q_r, D_r^+) - R(Q_r, D_{r,j}^-)$ (R is just cosine similarity)[75].
- $\mathbf{\Lambda}$ is all the parameter matrices (and biases) in the LSTM.

The authors then describe standard BPTT updates with momentum, which need not be detailed here. See the "Algorithm 1" figure in the paper for extremely detailed pseudo-code of the training procedure.

[75] Note that $\Delta_{r,j} \in [-2, 2]$. We use γ as a scaling factor so as to expand this range.

Clustering Massive Text Streams

Written by Brandon McKinzie

Aggarwal et al., "A Framework for Clustering Massive Text and Categorical Data Streams," (2006).

Overview. Authors present an online approach for clustering massive text and categorical data streams with the use of a statistical summarization methodology. First, we will go over the process of storing and maintaining the data structures necessary for the clustering algorithm. Then, we will discuss the differences which arise from using different kinds of data, and the empirical results.

Maintaining Cluster Statistics. The data stream consists of d-dimensional records, where each dimension corresponds to the numeric frequency of a given word in the vector space representation. Each data point is weighted by the **fading function** $f(t)$, a non-monotonic decreasing function which decays uniformly with time t. The authors define the **half-life** of a data point (e.g. a tweet) as:

$$t_0 \quad \text{s.t.} \quad f(t_0) = \frac{1}{2} f(0) \tag{249}$$

and, similarly, the **decay-rate** as its inverse, $\lambda = 1/t_0$. Thus we have $f(t) = 2^{-\lambda \cdot t}$.

To achieve greater accuracy in the clustering process, we require a high level of granularity in the underlying data structures. To do this, we will use a process in which condensed clusters of data points are maintained, referred to as **cluster droplets**. We define them differently for the case of text and categorical data, beginning with categorical:

- **Categorical**. A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of categorical data points $\mathcal{C}$ at time t is defined as the tuple:

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{250}$$

  where
  - Entry k of the vector $D\bar{F}2$ is the (weighted) number of points in cluster $\mathcal{C}$ where the ith dimension had value x and the j dimension had value y. In other words, all pairwise combinations of values in the categorical vector[76]. $\sum_{i=1}^{d} \sum_{j \neq i}^{d} v_i v_j$ entries total[77].
  - Similarly, $D\bar{F}1$ consists of the (weighted) counts that some dimension i took on the value x. $\sum_{i=1}^{d} v_i$ entries total.

[76] This is intentionally written hand-wavy because I'm really concerned with *text* streams and don't want to give this much space.

[77] v_i is the number of values the ith categorical dimension can take on.

- $w(t)$ is the sum of the weights of the data points at time t.
- l is the time stamp of the last time a data point was added to the cluster.

- **Text**. Can be viewed as an example of a sparse numeric data set. A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of text data points $\mathcal{C}$ at time t is defined as the tuple:

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{251}$$

where
- $D\bar{F}2$ contains $3 \cdot wb \cdot (wb - 1)/2$ entries, where wb is the number of distinct words in the *cluster* $\mathcal{C}$.
- $D\bar{F}1$ contains $2 \cdot wb$ entries.
- n is the number of data points in the cluster $\mathcal{C}$.

Cluster Droplet Maintenance.

1. We first start of with k trivial clusters (the first k data points that arrived).
2. When a new point $\bar{X}$ arrives, the cosine similarity to each cluster's $D\bar{F}1$ is computed.
3. $\bar{X}$ is inserted into the cluster for which this is a maximum, so long as the associated $S(\bar{X}, D\bar{F}1) >$ thresh, a predefined threshold. If not above the threshold *and* some **inactive cluster** exists, a new cluster is created containing the solitary point $\bar{X}$, which replaces the inactive cluster. If not above threshold but no inactive clusters, then we just insert it into the max similarity cluster anyway.
4. If $\bar{X}$ was inserted (i.e. didn't replace an inactive cluster), then we need to:
   (a) Update the statistics to reflect the decay of the data points at the current moment in time[78]. This is done by multiplying entries in the droplet vectors by $2^{-\lambda \cdot (t-l)}$.
   (b) Add the statistics for each newly arriving data point to the cluster statistics.

[78] In other words, the statistics for a cluster do not decay, until a new point is added to it.

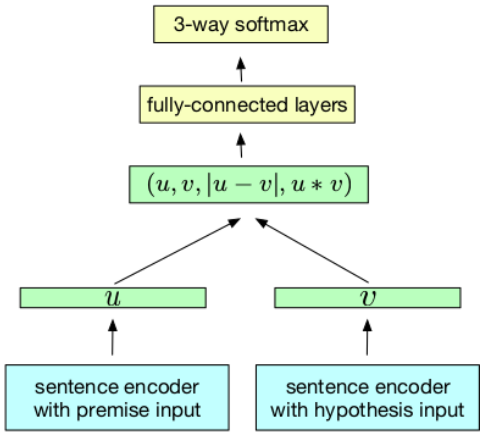
Supervised Universal Sentence Representations (InferSent)

 Written by Brandon McKinzie

Conneau et al., "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data," Facebook AI Research (2017).

Overview. Authors claim universal sentence representations trained using the supervised data of the Stanford Natural Language Inference (SNLI) dataset can consistently outperform unsupervised methods like SkipThought on a wide range of transfer tasks. They emphasize that training on NLI tasks in particular results in embeddings that perform well in transfer tasks. Their best encoder is a Bi-LSTM architecture with max pooling, which they claim is SOTA when trained on the SNLI data.

The Natural Language Inference Task. Also known as *Recognizing Textual Entailment* (RTE). The SNLI data consists of sentence pairs labeled as one of entailment, contradiction, or neutral. Below is a typical architecture for training on SNLI.



Note that the same sentence encoder is used for both u and v. To obtain a sentence vector from a BiLSTM encoder, they experiment with (1) the average h_t over all t (**mean pooling**), and (2) selecting the max value over each dimension of the hidden units [over all timesteps] (**max pooling**)[79].

[79] Since the authors have already mentioned that BiLSTM did the best, I won't go over the other architectures they tried: self-attentive networks, hierarchical convnet, vanilla LSTM/GRU.

Dist. Rep. of Sentences from Unlabeled Data (FastSent)

Written by Brandon McKinzie

Hill et al., "Learning Distributed Representations of Sentences from Unlabelled Data," (2016).

Overview. A systematic comparison of models that learn distributed representations of phrases/sentences from unlabeled data. Deeper, more complex models are preferable for representations to be used in supervised systems, but shallow log-linear models work best for building representation spaces that can be decoded with simple spatial distance metrics.

Authors propose two new phrase/sentence representation learning objectives:

1. **Sequential Denoising Autoencoders** (SDAEs)
2. **FastSent**: a sentence-level log-linear BOW model.

Distributed Sentence Representations. Existing models trained on text:

- **SkipThought Vectors** (Kiros et al., 2015). Predict target sentences $S_{i\pm1}$ given source sentence S_i. Sequence-to-sequence model.
- **ParagraphVector** (Le and Mikolov, 2014). Defines 2 log-linear models:
  1. **DBOW**: learns a vector $\boldsymbol{s}$ for every sentence S in the training corpus which, together with word embeddings v_w, define a softmax distribution to predict words $w \in S$ given S.
  2. **DM**: select k-grams of consecutive words $\{w_i \cdots w_{i+k} \in S\}$ and the sentence vector $\boldsymbol{s}$ to predict w_{i+k+1}.
- **Bottom-Up Methods**. Train **CBOW** and **Skip-Gram** word embeddings on the Books corpus.

The authors use **gensim** to implement ParagraphVector.

Models trained on *structured* (and freely-available) resources:

- **DictRep** (Hill et al., 2015a). Map dictionary definitions to pre-trained word embeddings, using either BOW or RNN-LSTM encoding.
- **NMT**. Consider sentence representations learned by sequence-to-sequence NMT models.

Novel Text-Based Methods.

- **Sequential (Denoising) Autoencoders**. To avoid needing coherent inter-sentence narrative, try this representation-learning objective based on DAEs. For a given sentence S and **noise function** $N(S \mid p_o, p_x)$ (where $p_0, p_x \in [0, 1]$),the approach is as follows:
  1. For each $w \in S$, N deletes w with probability p_o.
  2. For each non-overlapping bigram $w_i w_{i+1} \in S$, N swaps w_i and w_{i+1} with probability p_x.

     Authors recommend $p_o = p_x = 0.1$

     We then train the same LSTM-based encoder-decoder architecture as NMT, but with the denoising objective to predict (as target) the original source sentence S given a corrupted version $N(S \mid p_o, p_x)$ (as source).

- **FastSent**. Designed to be a more efficient/quicker to train version of SkipThought.

Latent Dirichlet Allocation

Written by Brandon McKinzie

Blei et al., "Latent Dirichlet Allocation," (2003).

Introduction. At minimum, one should be familiar with generative probabilistic models, mixture models, and the notion of latent variables before continuing. The "Dirichlet" in LDA of course refers to the **Dirichlet distribution**, which is a generalization of the beta distribution, B. It's PDF is defined as[80][81]:

$$\mathrm{Dir}(\boldsymbol{x};\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^{K}\boldsymbol{x}_i^{\boldsymbol{\alpha}_i-1} \quad \text{where} \quad B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K}\Gamma\left(\boldsymbol{\alpha}_i\right)}{\Gamma(\sum_{i=1}^{K}\boldsymbol{\alpha}_i)} \tag{252}$$

$$\sum_{i=1}^{K}\boldsymbol{x}_i = 1$$

$$(\forall i \in [1,K]) : \boldsymbol{x}_i \geq 0$$

Main things to remember about LDA:

- Generative probabilistic model for collections of discrete data such as text corpora.
- Three-level **hierarchical Bayesian model**. Each document is a mixture of topics, each topic is an infinite mixture over a set of topic probabilities.

Condensed comparisons/history of related models leading up to LDA:

- **TF-IDF**. Design matrix $\boldsymbol{X} \in \mathbb{R}^{V\times M}$, where M is the number of docs, and $\boldsymbol{X}_{i,j}$ gives the TF-IDF value for ith word in vocabulary and corresp. to document j.
- **LSI**:[82] Performs SVD on the TF-IDF design matrix $\boldsymbol{X}$ to identify a linear subspace in the space of tf-idf features that captures most of the variance in the collection.
- **pLSI**: **TODO**

> *pLSI is incomplete in that it provides no probabilistic model at the level of documents. In pLSI, each document is represented as a list of numbers (the mixing proportions for topics), and there is no generative probabilistic model for these numbers.*

[80] Recall that for positive integers n, $\Gamma(n) = (n-1)!$.

[81] The Dirichlet distribution is **conjugate** to the multinomial distribution. TODO: Review how to interpret this.

[82] Recall that LSI is basically PCA but without subtracting off the means

Model. LDA assumes the following generative process for each document (word sequence) $\boldsymbol{w}$:

1. $N \sim \textbf{Poisson}(\lambda)$: Sample N, the number of words (length of $\boldsymbol{w}$), from $\text{Poisson}(\lambda) = e^{-\lambda}\frac{\lambda^n}{n!}$. The parameter λ *should* represent the average number of words per document.
2. $\theta \sim \textbf{Dir}(\alpha)$: Sample k-dimensional vector θ from the Dirichlet distribution (eq. 252), $\text{Dir}(\alpha)$. k is the number of topics (pre-defined by us). Recall that this means θ lies in the (k-1) simplex. The Dirichlet distribution thus tells us the probability density of θ over this simplex – it defines the probability of θ being at a given position on the simplex.
3. Do the following N times to generate the words for this document.
   (a) $z_n \sim \textbf{Multinomial}(\theta)$. Sample a topic z_n.
   (b) $w_n \sim \textbf{Pr}\left[w_n \mid z_n, \beta\right]$: Sample a word w_n from $\Pr\left[w_n \mid z_n, \beta\right]$, a "multinomial probability conditioned on topic z_n."[83] The parameter β gives the distribution of words given a topic:

$$\beta_{ij} = \Pr\left[w_j \mid z_i\right] \tag{253}$$

   In other words, we really sample $w_n \sim \beta_{i,:}$

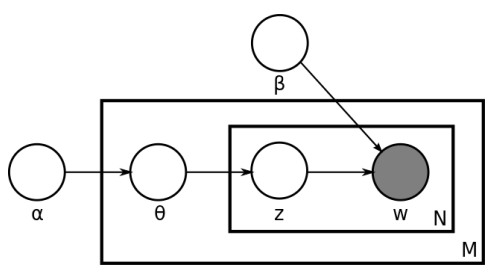
The defining equations for LDA are thus:

$$\Pr\left[\theta, \boldsymbol{z}, \boldsymbol{w} \mid \alpha, \beta\right] = \textbf{Pr}\left[\theta \mid \alpha\right] \prod_{n=1}^{N} \textbf{Pr}\left[z_n \mid \theta\right] \textbf{Pr}\left[w_n \mid z_n, \beta\right] \tag{254}$$

$$\Pr\left[\boldsymbol{w} \mid \alpha, \beta\right] = \int \textbf{Pr}\left[\theta' \mid \alpha\right] \left(\prod_{n=1}^{N} \sum_{z'_n} \textbf{Pr}\left[z'_n \mid \theta'\right] \textbf{Pr}\left[w_n \mid z'_n, \beta\right] \right) \mathrm{d}\theta' \tag{255}$$

$$\Pr\left[\mathcal{D} = \{\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(M)}\} \mid \alpha, \beta\right] = \prod_{d=1}^{M} \Pr\left[\boldsymbol{w}^{(d)} \mid \alpha, \beta\right] \tag{256}$$

Below is the plate notation for LDA, followed by an interpretation:



- **Outermost Variables**: α and β. Both represent a (Dirichlet) prior distribution: α parameterizes the probability of a given *topic*, while β a given *word*.
- Document Plate. M is the number of documents, $\boldsymbol{\theta}_m$ gives the true distribution of topics for document m[84].

[83] **TODO**: interpret meaning of the multinomial distributions here. Seems a bit different than standard interp...

[84] In other words, the meaning of $\boldsymbol{\theta}_{m,i} = x$ is "x percent of document m is about topic i."

- Topic/Word Place. $\boldsymbol{z}_{mn}$ is the topic for word n in doc m, and $\boldsymbol{w}_{mn}$ is the word. It is shaded gray to indicate it is the only **observed variable**, while all others are **latent variables**.

Theory. I'll quickly summarize and interpret the main theoretical points. Without having read all the details, this won't be of much use (i.e. it is for someone who has read the paper already).

- **LDA and Exchangeability**. We assume that each document is a bag of words (order doesn't matter; frequency sitll does) *and* a bag of topics. In other words, a document of N words *is* an unordered list of words and topics. De Finetti's theorem tells us that we can model the joint probability of the words and topics as if a random parameter θ were drawn from some distribution and then the variables within $\boldsymbol{w}, \boldsymbol{z}$ were **conditionally independent given** θ. LDA posits that a good distribution to sample θ from is a Dirichlet distribution.
- **Geometric Interpretation**: **TODO**

Inference and Parameter Estimation. As usual, we need to find a way to compute the posterior distribution of the hidden variables given a document $\boldsymbol{w}$:

$$\Pr\left[\boldsymbol{\theta}, \boldsymbol{z} \mid \boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right] = \frac{\Pr\left[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]}{\Pr\left[\boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]} \tag{257}$$

Computing the denominator exactly is intractable. Common approximate inference algorithms for LDA include Laplace approximation, variational approximation, and Markov Chain Monte Carlo.

Conditional Random Fields

Written by Brandon McKinzie

Lafferty et al., "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," (2001).

Introduction. CRFs offer improvements to HMMs, MEMMs, and other discriminative Markov models. MEMMs and other non-generative models share a weakness called the **label bias problem**: the transitions leaving a given state compete only against each other, rather than against all other transitions in the model. The key difference between CRFs and MEMMs is that a CRF has a single exponential model for the joint probability of the entire sequence of labels given the observation sequence.

The Label Bias Problem. Recall that MEMMs are run left-to-right. One way of interpreting such a model is to consider how the probabilities (of state sequences) are distributed as we continue through the sequence of observations. The issue with MEMMs is that there's nothing we can do if, somewhere along the way, we observe something that makes one of these state paths extremely likely/unlikely; we can't redistribute the probability mass amongst the various allowed paths. The CRF solution:

> *Account for whole state sequences at once by letting some transitions "vote" more strongly than others depending on the corresponding observations. This implies that score mass will not be conserved, but instead individual transitions can "amplify" or "dampen" the mass they receive.*

Conditional Random Fields. Here we formalize the model and notation. Let $\mathbf{X}$ be a random variable over data sequences to be labeled (e.g. over all words/sentences), and let $\mathbf{Y}$ the random variable over corresponding label sequences[85]. Formal definition:

> *Let $G = (V, E)$ be a graph such that $Y = (Y_v)_{v \in V}$, so that Y is indexed by the vertices of G. Then (X, Y) is a CRF if, when conditioned on X, the random variables Y_v obey the Markov property with respect to the graph:*
>
> $$Pr\left[Y_v | X, Y_w, w \neq v\right] = Pr\left[Y_v | X, Y_w, w \sim v\right] \tag{258}$$
>
> *where $w \sim v$ means that w and v are neighbors in G.*

All this means is a CRF is a random field (discrete set of random-valued points in a space) where all points (i.e. globally) are conditioned on $\mathbf{X}$. If the graph $G = (V, E)$ of $\mathbf{Y}$ is a tree, its cliques[86] are the edges and vertices. Take note that $\mathbf{X}$ is not a member of the vertices

[85] We assume all components $\mathbf{Y}_i$ can only take on values in some finite label set $\mathcal{Y}$.

[86] A clique is a subset of vertices in an undirected graph such that every two distinct vertices in the clique are adjacent

in G. G only contains vertices corresponding to elements of $\mathbf{Y}$. Accordingly, when the authors refer to cases where G is a "chain", remember that they just mean the $\mathbf{Y}$ vertex sequence.

By the fundamental theorem of random fields:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \propto \exp\left(\sum_{e \in E,k} \lambda_k f_k(e, \mathbf{y}|_e, \mathbf{x}) + \sum_{v \in V,k} \mu_k g_k(v, \mathbf{y}|_v, \mathbf{x})\right) \tag{259}$$

where $\mathbf{y}|_S$ is the set of components of $\mathbf{y}$ associated with the vertices in subgraph S. We assume the K feature [functions] f_k and g_k are given and fixed. Note that f_k are the feature functions over *transitions* y_{t-1} to y_t, and g_k are the feature functions over *states* y_t and x_t. Our estimation problem is thus to determine parameters $\theta = (\lambda_1, \lambda_2, \ldots; \mu_1, \mu_2, \ldots)$ from the labeled training data.

Linear-Chain CRF. Let $|\mathcal{Y}|$ denote the number of possible labels. At each position t in the observation sequence $\boldsymbol{x}$, we define the $|\mathcal{Y}| \times |\mathcal{Y}|$ matrix random variable $\boldsymbol{M}_t(\boldsymbol{x})$

$$\boldsymbol{M}_t(y', y, \mid \boldsymbol{x}) = \exp(\boldsymbol{\Lambda}_t(y', y \mid x)) \tag{260}$$

$$\boldsymbol{\Lambda}_t(y', y \mid x) = \sum_k \lambda_k f_k(y', y, \boldsymbol{x}) + \sum_k \mu_k g_k(y, \boldsymbol{x}) \tag{261}$$

where $y_{t-1} := y'$ and $y_t := y$. We can see that the individual elements correspond to specific values of e and v in the double-summations of $p_\theta(\boldsymbol{y} \mid \boldsymbol{x})$ above. Then the normalization (partition function) $Z_\theta(\boldsymbol{x})$ is the (y_0, y_{T+1}) entry (the fixed boundary states) of the product:

$$Z_\theta(\boldsymbol{x}) = \left[\prod_{t=1}^{T+1} \boldsymbol{M}_t(\boldsymbol{x})\right]_{y_0, y_{T+1}} \tag{262}$$

which includes all possible sequences $\boldsymbol{y}$ that start with the fixed y_0 and end with the fixed y_{T+1}. Now we can write the conditional probability as a function of just these matrices:

$$p_\theta(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\prod_{t=1}^{T+1} \boldsymbol{M}_t(y_{t-1}, y_t \mid \boldsymbol{x})}{\left[\prod_{t=1}^{T+1} \boldsymbol{M}_t(\boldsymbol{x})\right]_{y_0, y_{T+1}}} \tag{263}$$

Parameter Estimation (for linear-chain CRFs). For each t in $[0, T+1]$, define the **forward vectors** $\alpha_t(\boldsymbol{x})$ with base case $\alpha_0(y \mid \boldsymbol{x}) = 1$ if $y = y_0$, else 0. Similarly, define the **backward vectors** $\beta_t(\boldsymbol{x})$ with base case $\beta_{T+1}(y \mid \boldsymbol{x}) = 1$ if $y = y_{T+1}$ else 0[87]. Their recurrence relations are

$$\alpha_t(\boldsymbol{x}) = \alpha_{t-1}(\boldsymbol{x})\boldsymbol{M}_t(\boldsymbol{x}) \tag{264}$$

$$\beta_t(\boldsymbol{x})^T = \boldsymbol{M}_{t+1}(\boldsymbol{x})\beta_{t+1}(\boldsymbol{x}) \tag{265}$$

[87] Remember that y_0 and y_{T+1} are their own fixed symbolic constants representing a fixed start/stop state.

Attention Is All You Need

Written by Brandon McKinzie

Vaswani et al., "Attention Is All You Need," (2017)

Overview. Authors refer to sequence *transduction* models a lot – just a fancy way of referring to models that transform input sequences into output sequences. Authors propose new architecture, the **Transformer**, based solely on attention mechanisms (no recurrence!).

Model Architecture.

- **Encoder**. N = 6 identical layers, each with 2 sublayers: (1) a **multi-head self-attention** mechanism and (2) a position-wise FC feed-forward network. They apply a residual connection and layer norm such that each sublayer, instead of outputting Sublayer(x), instead outputs LayerNorm(x + Sublayer(x)).
- **Decoder**. N = 6 with 3 sublayers each. In addition to the two sublayers described for the encoder, the decoder has a third sublayer, which performs **multi-head** attention over the output of the encoder stack. Same residual connections and layer norm.

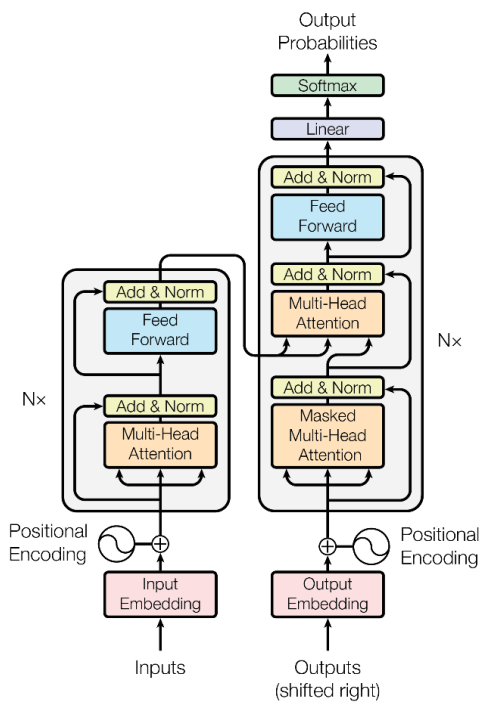


Figure shows encoder-decoder template layers. The actual model instantiates chain of 6 encoder layers and 6 decoders layers. The decoder's self-attention masks embeddings at future timesteps to zero.

Attention. An attention function can be described as a mapping:

$$\text{Attn}(\text{query}, \{(k_1, v_1), \ldots, \}) \Rightarrow \sum_i fn(\text{query}, k_i)v_i \tag{266}$$

where the query, keys, values, and output are all vectors.

- **Scaled Dot-Product Attention**.
  1. **Inputs**: queries q, keys k of dimension d_k, values v of dimension d_v
  2. **Dot Products**: Compute $\forall k: \ (q \cdot k)/\sqrt{d_k}$.
  3. **Softmax**: on each dot product above. This gives the weights on the values shown earlier.

  Appears that $d_q \equiv d_k$.

  In practice, this is done simultaneously for all queries in a set via the following matrix equation:

  $$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{267}$$

  Note that this is identical to the standard dot-product attention mechanism, except for the *scaling* factor (hence the name) of $1/\sqrt{d_k}$. The scaling factor is motivated by the fact that additive attention outperforms dot-product attention for large d_k and the authors stipulate this is due to the softmax having small gradients in this case, due to the large dot products[88].
- **Multi-Head Attention**. Basically just doing some number h of parallel attention computations. Before each of these, the queries, keys, and values are linearly projected with different, learned linear projections to d_k, d_k and d_v dimensions respectively (and then fed to their respective attention function). The h outputs are then concatenated and once again projected, resulting in the final values.

  $$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{268}$$

  $$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{269}$$

  $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$

  $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$

  $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$

  $W^O \in \mathbb{R}^{hd_v \times d_{model}}$

  The authors employ $h = 8$, $d_k = d_v = d_{model}/h = 64$.

The Transformer uses multi-headed attention in 3 ways:

1. **Encoder-decoder attention**: the normal kind. Queries are previous decoder layer, and memory keys and values come from output of the [final layer of] encoder.
2. **Encoder self-attention**: all of the keys, values, and queries come from the previous *layer* in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
3. **Decoder self-attention**: Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position (timestep). The masking is done on the inputs to the softmax, setting all inputs beyond the current timestep to $-\infty$.

[88] Thus, they insert the $1/\sqrt{d_k}$ to counteract this effect.

Other Components.

- **Position-wise Feed-Forward Networks (FFN)**: each layer of the encoder and decoder contains a FC FFN, applied to each position separately and identically:

$$\mathrm{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2 \tag{270}$$

The FFN is linear → ReLU → linear.

- **Embeddings and Softmax**: use learned embeddings to convert input/output tokens to vectors of dimension d_{model}, *and* for the pre-softmax layer at the output of the decoder[89].

For inputs to encoder/decoder, the embedding weights are multiplied by $\sqrt{d_{model}}$

- **Positional Encoding**: how the authors deal with the lack of recurrence (to make use of the sequence order). They add a sinusoid function of the position (timestep) *pos* and vector index i to the input embeddings for the encoder and decoder[90]:

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{271}$$

$$\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{272}$$

The authors justify this choice:

> *We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.*

[89] In other words, they use the same weight matrix for all three of (1) encoder input embedding, (2) decoder input embedding, and (3) (opposite direction) from decoder output to pre-softmax.

[90] Note that the positional encodings must necessarily be of dimension d_{model} to be summed with the input embeddings.

Hierarchical Attention Networks

Written by Brandon McKinzie

Yang et al., "Hierarchical Attention Networks for Document Classification."

Overview. Authors introduce the Hierarchical Attention Network (HAN) that is designed to capture insights regarding (1) the hierarchical structure of documents (words -> sentences -> documents), and (2) the context dependence between words and sentences. The latter is implemented by including two levels of attention mechanisms, one at the word level and one at the sentence level.

Hierarchical Attention Networks. Below is an illustration of the network. The first stage is familiar to sequence to sequence models - a bidirectional encoder for outputting sentence-level representations of a sequence of words. The HAN goes a step further by feeding this another bidirectional encoder for outputting document-level representations for sequences of sentences.

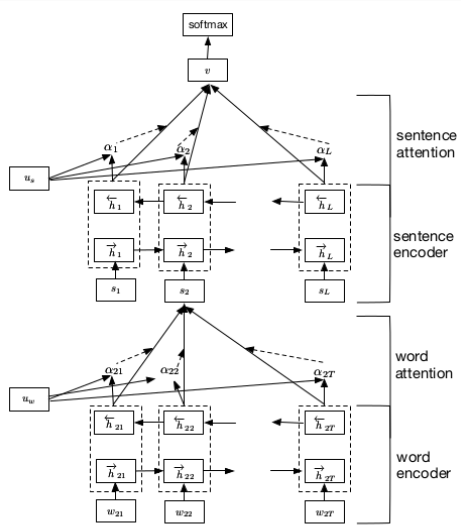


Figure 2: Hierarchical Attention Network.

The authors choose the GRU as their underlying RNN. For ease of reference, the defining equations of the GRU are shown below:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{273}$$

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \tag{274}$$

$$\tilde{h}_t = \tanh\left(W_h x_t + r_t \odot (U_h h_{t-1}) + b_h\right) \tag{275}$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{276}$$

Hierarchical Attention. Here I'll overview the main stages of information flow.

1. **Word Encoder**. Let the tth word in the ith sentence be denoted w_{it}. They embed the vectors with a word embedding matrix W_e, $x_{it} = W_e w_{it}$, and then feed x_{it} through a bidirectional GRU to ultimately obtain $h_{it} := [\overrightarrow{h}_{it}; \overleftarrow{h_{it}}]$.
2. **Word Attention**. Extracts words that are important to the meaning of the sentence and aggregates the representation of these informative words to form a sentence vector.

$$u_{it} = \tanh(W_w h_{it} + b_w) \tag{277}$$

$$\alpha_{it} = \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)} \tag{278}$$

$$s_i = \sum_t \alpha_{it} h_{it} \tag{279}$$

   Note the *context vector* u_w, which is shared for all words[91] and randomly initialized and jointly learned during the training process.
3. **Sentence Encoder**. Similar to the word encoder, but uses the sentence vectors s_i as the input for the ith sentence in the document. Note that the output of this encoder, h_i contains information from the neighboring sentences too (bidirectional) but focuses on sentence i.
4. **Sentence Attention**. For rewarding sentences that are clues to correctly classify a document. Similar to before, we now use a sentence level context vector u_s to measure the importance of the sentences.

$$u_i = \tanh(W_s h_i + b_s) \tag{280}$$

$$\alpha_i = \frac{\exp(u_i^T u_s)}{\sum_i \exp(u_i^T u_s)} \tag{281}$$

$$v = \sum_t \alpha_i h_i \tag{282}$$

   where v is the document vector that summarizes all the information of sentences in a document.

As usual, we convert v to a normalized probability vector by feeding through a softmax:

$$p = \text{softmax}(W_c v + b_c) \tag{283}$$

[91] To emphasize, there is only a single context vector u_w in the network, period. The subscript just tells us that it is the word-level context vector, to distinguish it from the sentence-level context vector in the later stage.

Configuration and Training. Quick overview of some parameters chosen by the authors:

- **Tokenization**: Stanford CoreNLP. Vocabulary consists of words occurring more than 5 times, all others are replaced with UNK token.
- **Word Embeddings**: train word2vec on the training and validation splits. Dimension of 200.
- **GRU**. Dimension of 50 (so 100 because bidirectional).
- **Context vectors**. Both u_w and u_s have dimension of 100.
- **Training**: batch size of 64, grouping documents of similar length into a batch. SGD with momentum of 0.9.

Joint Event Extraction via RNNs

Written by Brandon McKinzie

Nguyen, Cho, and Grishman, "Joint Event Extraction via Recurrent Neural Networks," (2016).

Event Extraction Task. Automatic Context Extraction (ACE) evaluation. Terminology:

- **Event**: something that happens or leads to some change of state.
- **Mention**: phrase or sentence in which an event occurs, including one trigger and an arbitrary number of arguments.
- **Trigger**: main word that most clearly expresses an event occurrence.
- **Argument**: an entity mention, temporal expression, or value that serves as a participant/attribute with a specific role in an event mention.

Example:

> In Baghdad, a **cameraman** **died**{*Die*} when an American tank **fired**{*Attack*} on the Palestine hotel.

TRIGGER
ARGUMENT

Each event subtype has its own set of roles to be filled by the event arguments. For example, the roles for the *Die* event subtype include *Place*, *Victim*, and *Time*.

Model.

- **Sentence Encoding**. Let w_i denote the ith token in a sentence. It is transformed into a real-valued vector x_i, defined as

$$x_i := [\text{GloVe}(w_i);\ \text{Embed}(\text{EntityType}(w_i));\ \text{DepVec}(w_i)] \tag{284}$$

  where "Embed" is an embedding we learn, and "DepVec" is the binary vector whose dimensions correspond to the possible relations between words in the dependency trees.
- **RNN**. Bidirectional LSTM on the inputs x_i.
- **Prediction**. Binary memory vector G_i^{trg} for triggers; binary memory matrices G_i^{arg} and $G_i^{arg/trg}$ for arguments (at each timestep i). At each time step i, do the following in order:
  1. Predict trigger t_i for w_i. First compute the feature representation vector R_i^{trig}, defined as:

$$R_i^{trig} := \left[h_i;\ L_i^{trg};\ G_{i-1}^{trg}\right] \tag{285}$$

     where h_i is the RNN output, L_i^{trg} is the local context vector for w_i, and G_{i-1}^{trg} is the memory vector from the previous step. $L_i^{trg} := [\text{GloVe}(w_{i-d}); \ldots ; \text{GloVe}(w_{i+d})]$ for

some predefined window size d. This is then fed to a fully-connected layer with softmax activation, $\boldsymbol{F}^{trg}$, to compute the probability over possible trigger subtypes:

$$P_{i;t}^{trg} := F_t^{trg}(R_i^{trg}) \tag{286}$$

As usual, the predicted trigger type for w_i is computed as $t_i = \arg\max_t \left(P_{i;t}^{trg}\right)$. If w_i is not a trigger, t_i should predict "*Other*."

2. Argument role predictions, $a_{i1}, \ldots, a_{ik}$, for all of the [already known] entity mentions in the sentence, $e_1, \ldots, e_k$ with respect to w_i. a_{ij} denotes the argument role of e_j with respect to [the predicted trigger of] w_i. If NOT(w_i is trigger AND e_j is one of its arguments), then a_{ij} is set to *Other*. For example, if w_i was the word "died" from our example sentence, we'd hope that its predicted trigger would be $t_i = Die$, and that **the entity associated with "cameraman" would get a predicted argument role of *Victim*.**

```
def getArgumentRoles(triggerType=t, entities=e):
   k = len(e)
   if isOther(t):
      return [Other] * k
   else:
      for e_j in e:
```

$$R_{ij}^{arg} := \left[h_i;\ h_{i_j};\ L_{ij}^{arg};\ B_{ij};\ G_{i-1}^{arg}[j]; G_{i-1}^{arg/trg}[j]\right] \tag{287}$$

3. Update memory. TO BE CONTINUED...(moving onto another paper because this model is getting a *bit* too contrived for my tastes. Also not a fan of the reliance on a dependency parse.)

Event Extraction via Bidi-LSTM Tensor NNs

Written by Brandon McKinzie

Y. Chen, S. Liu, S. He, K. Liu, and J. Zhao, "Event Extraction via Bidirectional Long Short-Term Memory Tensor Neural Networks."

Overview. The task/goal is the event extraction task as defined in *Automatic Content Extraction* (ACE). Specifically, given a text document, our goal is to do the following in order *for each sentence*:

1. Identify any event triggers in the sentence.
2. If triggers found, predict their subtype. For example, given the trigger "fired," we may classify it as having the *Attack* subtype.
3. If triggers found, identify their candidate argument(s). ACE defines an event argument as "an entity mention, temporal expression, or value that is involved in an event."
4. For each candidate argument, predict its role: "the relationship between an argument to the event in which it participates."

Context-aware Word Representation. Use pre-trained word embeddings for the input word tokens, the predicted trigger, and the candidate argument. Note: *we assume we already have predictions for the event trigger t and are doing a pass for one of (possibly many) candidate arguments a.*

1. Embed each word in the sentence with pre-trained embeddings. Denote the embedding for ith word as $e(w_i)$.
2. Feed each $e(w_i)$ through a bidirectional LSTM. Denote the ith output of the forward LSTM as $c_l(w_{i+1})$ and the output of the backward LSTM at the same time step as $c_r(w_{i-1})$. As usual, they take the general functional form:

$$c_l(w_i) = \overrightarrow{LSTM}\left(c_l(w_{i-1}), e(w_{i-1})\right) \tag{288}$$

$$c_r(w_i) = \overleftarrow{LSTM}\left(c_r(w_{i+1}), e(w_{i+1})\right) \tag{289}$$

$$\tag{290}$$

3. Concatenate $e(w_i)$, $c_l(w_i)$, $c_r(w_i)$ together along with the embedding of the candidate argument $e(a)$ and predicted trigger $e(t)$. Also include the relative distance of w_i to t or (??) a, denoted as pi for position information, and the embedding of the predicted event type pe of the trigger. Denote this massive concatenation result as x_i:

$$x_i := c_l(w_i) \oplus e(w_i) \oplus c_r(w_i) \oplus pi \oplus pe \oplus e(a) \oplus e(t) \tag{291}$$

Dynamic Multi-Pooling. This is easiest shown by example. Continue with our example sentence:

> In California, **Peterson** was arrested for the **murder** of his wife and unborn son.

where the colors are given for *this specific case where murder is our predicted trigger and we are considering the candidate argument Peterson*[92]. Given our n outputs from the previous stage, $y^{(1)} \in \mathbb{R}^{n \times m}$, where n is the length of the sentence and m is the size of that huge concatenation given in equation 291. We split our sentence by trigger and candidate argument, then (confusingly) redefine our notation as

$$y_{1j}^{(1)} \leftarrow \begin{bmatrix} y_{1j}^{(1)} & y_{2j}^{(1)} \end{bmatrix} \tag{292}$$

$$y_{2j}^{(1)} \leftarrow \begin{bmatrix} y_{3j}^{(1)} & \cdots & y_{7j}^{(1)} \end{bmatrix} \tag{293}$$

$$y_{3j}^{(1)} \leftarrow \begin{bmatrix} y_{8j}^{(1)} & \cdots & y_{nj}^{(1)} \end{bmatrix} \tag{294}$$

Peterson is the 3rd word, and murder is the 8th word.

where it's important to see that, for some $1 \leq j \leq m$, each new $y_{ij}^{(1)}$ is a *vector* of length equal to the number of words in segment i. Finally, the dynamic multi-pooling layer, $y^{(2)}$, can be expressed as

$$y_{i,j}^{(2)} := \max\left(y_{i,j}^{(1)}\right) \qquad 1 \leq i \leq 3,\ 1 \leq j \leq m \tag{295}$$

where the max is taken over each of the aforementioned vectors, leaving us with $3m$ values total. These are concatenated to form $y^{(2)} \in \mathbb{R}^{3m}$.

Output. To predict of each argument role [for the given argument candidate], $y^{(2)}$ is fed through a dense softmax layer,

$$O = W_2 y^{(2)} + b_2 \tag{296}$$

where $W_2 \in \mathbb{R}^{n_1 \times 3m}$ and n_1 is the number of possible argument roles (including "None"). The authors also use dropout on $y^{(2)}$.

[92] Yes, arrested could be another predicted trigger, but the network considers each possibility at separate times/locations in the architecture.

Reasoning with Neural Tensor Networks

Written by Brandon McKinzie

Socher et al., "Reasoning with Neural Tensor Networks for Knowledge Base Completion"

Overview. Reasoning over relationships between two entities. Goal: predict the likely truth of additional facts based on existing facts in the KB. This paper contributes (1) the new NTN and (2) a new way to represent entities in KBs. Each relation is associated with a distinct model. Inputs to a given relation's model are pairs of database entities, and the outputs score how likely the pair has the relationship.

Neural Tensor Networks for Relation Classification. Let $e_1, e_2 \in \mathbb{R}^d$ be the vector representations of the two entities, and let R denote the relation (and thus model) of interest. The NTN computes a score of how likely it is that e_1 and e_2 are related by R via:

$$g(e_1, R, e_2) = u_R^T \tanh\left(e_1^T W_R^{[1:k]} e_2 + V_R \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b_R\right) \tag{297}$$

$W_R^{[1:k]} \in \mathbb{R}^{d \times d \times k}$

$V_R \in \mathbb{R}^{k \times 2d}$

where the bilinear tensor product $e_1^T W_R^{[1:k]} e_2$ results in a vector $h \in \mathbb{R}^k$ with each entry computed by one slice $i = 1, \ldots, k$ of the tensor.

> *Intuitively, we can see each slice of the tensor as being responsible for one type of entity pair or instantiation of a relation... Another way to interpret each tensor slice is that it mediates the relationship between the two entity vectors differently.*

Training Objective and Derivatives. All models are trained with **contrastive max-margin objective functions** and minimize the following objective:

$$J(\mathbf{\Omega}) = \sum_{i=1}^{N} \sum_{c=1}^{C} \max\left(0, 1 - g\left(T^{(i)}\right) + g\left(T_c^{(i)}\right)\right) + \lambda ||\mathbf{\Omega}||_2^2 \tag{298}$$

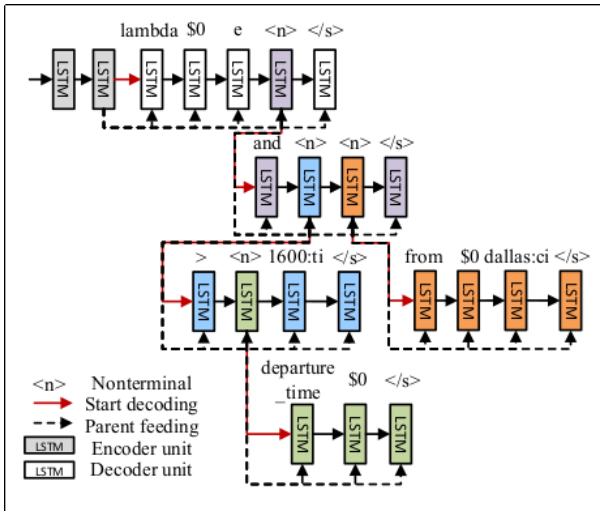
where c is for "corrupted" samples, $T_c^{(i)} := \left(e_1^{(i)}, R^{(i)}, e_c^{(i)}\right)$. Notice that this function is minimized when the difference, $g\left(T^{(i)}\right) - g\left(T_c^{(i)}\right)$, is maximized. The authors used minibatched **L-BFGS** for optimization.

Language to Logical Form with Neural Attention

 Written by Brandon McKinzie

Dong and Lapata, "Language to Logical Form with Neural Attention," (2016)

Sequence-to-Tree Model. Variant of Seq2Seq that is more faithful to the compositional nature of meaning representations. It's schematic is shown below. The authors define a "nonterminal" $< n >$ token which indicates [the root of] a subtree.



where the author's have employed "parent-feeding": for a given subtree (logical form), at each timestep, the hidden vector of the parent nonterminal is concatenated with the inputs and fed into the LSTM (best understood via above illustration).

> *After encoding input q, the hierarchical tree decoder generates tokens at depth 1 of the subtree corresponding to parts of logical form a. If the predicted token is $< n >$, decode the sequence by conditioning on the nonterminal's hidden vector. This process terminates when no more nonterminals are emitted.*

Also note that the output posterior probability over the encoded input q is the product of subtree posteriors. For example, consider the decoding example in the figure below:

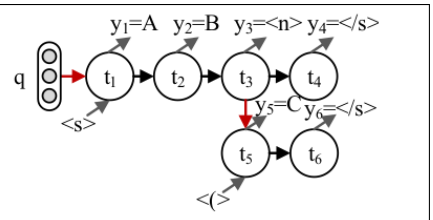


Figure 4: A SEQ2TREE decoding example for the logical form "*A B (C)*".

We would compute the output posterior as:

$$p(a \mid q) = p(y_1 y_2 y_3 y_4 \mid q) p(y_5 y_6 \mid y_{\leq 3}, q) \tag{299}$$

The model is trained by minimizing log-likelihood over the training data, using RMSProp for optimization. At inference time, greedy search or beam search is used to predict the most probable output sequence.

Seq2SQL: Generating Structured Queries from NL using RL

Zhong, Xiong, and Socher, "Seq2SQL: Generating Structured Queries From Natural Language Using Reinforcement Learning"

Overview. Deep neural network for translating natural language questions to corresponding SQL queries. Outperforms state-of-the-art semantic parser.

Seq2Tree and Pointer Baseline. Baseline model is the Seq2Tree model from the previous note on Dong & Lapata's (2016) paper. Authors here argue their output space is unnecessarily large, and employ the idea of pointer networks with augmented inputs. The input sequence is the concatenation of (1) the column names, (2) the limited vocabulary of the SQL language such as `SELECT`, `COUNT`, etc., and (3) the question.

$$x := [< col >;\ x_1^c; x_2^c; \ldots; x_N^c;\ \ < sql >;\ x^s;\ \ < question >;\ x^q] \tag{300}$$

$x_j^c \in \mathbb{R}^{T_j}$

where we also insert special ("sentinel") tokens to demarcate the boundaries of each section. The pointer network can then produce the SQL query by selecting exclusively from the input. Let g_s denote the sth decoder [hidden] state, and y_s denote the output (index/pointer to input query token).

$$\textbf{[ptr net]}\ y_s = \arg\max_t \left(\alpha_s^{ptr}\right) \quad \text{where} \quad \alpha_{s,t}^{ptr} = w^{ptr} \cdot \tanh\left(U^{ptr} g_s + V^{ptr} h_t\right) \tag{301}$$

Seq2SQL.

1. **Aggregation Classifier**. Our goal here is to predict which aggregation operation to use out of `COUNT`, `MIN`, `MAX`, `NULL`, etc. This is done by projecting the attention-weighted average of encoder states, κ^{agg}, to $\mathbb{R}^C$ where C denotes the number of unique aforementioned aggregation operations. The sequence of computations is summarized as follows:

$$\alpha_t^{inp} = w^{inp} \cdot h_t^{enc} \tag{302}$$

$$\beta^{inp} = \text{softmax}\left(\alpha^{inp}\right) \tag{303}$$

$W^{agg} \in \mathbb{R}^{C \times T}$

$\beta^{agg} \in \mathbb{R}^C$

$$\kappa^{agg} = \sum_t^T \beta_t^{inp} h_t^{enc} \tag{304}$$

$$\alpha^{agg} = W^{agg} \tanh\left(V^{agg} \kappa^{agg} + b^{agg}\right) + c^{agg} \tag{305}$$

$$\beta^{agg} = \text{softmax}\left(\alpha^{agg}\right) \tag{306}$$

where β_i^{agg} gives the probability for the ith aggregation operation. We use cross entropy loss L^{agg} for determining the aggregation operation. Note that this part isn't really a sequence-to-sequence architecture. It's nothing more than an MLP applied to an attention-weighted average of the encoder states.

2. **Get Pointer to Column**. A pointer network is used for identifying which column in the input representation should be used in the query. Recall that $x_{j,t}^c$ denotes the tth word in column j. We use the last encoder state for a given column's LSTM[93] as its representation; T_j denotes the number of words in the jth column.

$$e_j^c = h_{j,T_j}^c \quad \text{where} \quad h_{j,t}^c = \text{LSTM}\left(\text{emb}(x_{j,t}^c), h_{j,t-1}^c\right) \tag{307}$$

To construct a representation for the question, compute another input representation κ^{sel} using the same architecture (but distinct weights) as for κ^{agg}. As usual, we compute the scores for each column j via:

$$\alpha_j^{sel} = W^{sel} \tanh\left(V^{sel}\kappa^{sel} + V^c e_j^c\right) \tag{308}$$

$$\beta^{sel} = \text{softmax}\left(\alpha^{sel}\right) \tag{309}$$

Similar to the aggregation, we train the `SELECT` network using cross entropy loss L^{sel}.

3. **WHERE Clause Pointer Decoder**. Recall from equation 301 that this is a model with recurrent connections from its *outputs leading back into its inputs*, and thus a common approach is to train it with **teacher forcing**[94]. However, since the boolean expressions within a `WHERE` clause can be swapped around while still yielding the same SQL query, reinforcement learning (instead of cross entropy) is used to *learn a policy to directly optimize the expected correctness of the execution result*. Note that this also implies that we will be sampling from the output distribution at decoding step s to obtain the next input for $s+1$ [instead of teacher forcing].

[93] Yes, we encode each column with an LSTM separately.

[94] Teacher forcing is just a name for how we train the decoder portion of a sequence-to-sequence model, wherein we feed the ground-truth output $y^{(t)}$ as input at time $t+1$ during training.

$$R\left(q(y), q_g\right) = \begin{cases} -2 & \text{if } q(y) \text{ is not a valid SQL query} \\ -1 & \text{if } q(y) \text{ is a valid SQL query and executes to an incorrect result} \\ +1 & \text{if } q(y) \text{ is a valid SQL query and executes to the correct result} \end{cases} \tag{310}$$

$$L^{whe} = -\mathbb{E}_y\left[R\left(q(y), q_g\right)\right] \tag{311}$$

$$\nabla L_{\Theta}^{whe} = -\nabla_{\Theta}\left(\mathbb{E}_{y\sim p_y}\left[R\left(q(y), q_g\right)\right]\right) \tag{312}$$

$$= -\mathbb{E}_{y\sim p_y}\left[R\left(q(y), q_g\right)\nabla_{\Theta}\sum_t \log p_y(y_t;\Theta)\right] \tag{313}$$

$$\approx -R\left(q(y), q_g\right)\nabla_{\Theta}\sum_t \log p_y(y_t;\Theta) \tag{314}$$

where

→ $y = [y^1, y^2, \ldots, y^T]$ denotes the sequences of generated tokens in the `WHERE` clause.
→ $q(y)$ denotes the query generated by the model.
→ q_g denotes the ground truth query corresponding to the question.

and the gradient has been approximated in the last line using a single Monte-Carlo sample y.

Finally, the model is trained using gradient descent to minimize $L = L^{agg} + L^{sel} + L^{whe}$.

Speculations for Event Extraction. I want to explore using this paper's model for the task of event extraction. Below, I've replaced some words (shown in green) from a sentence in the paper in order to formalize this as event extraction.

> *Seq2**Event** takes as input a **sentence** and the **possible event types** of an **ontology**. It generates the corresponding **event annotation**, which, during training, is **compared** against an **event template**. The result of the **comparison** is utilized to train the reinforcement learning algorithm*[95].

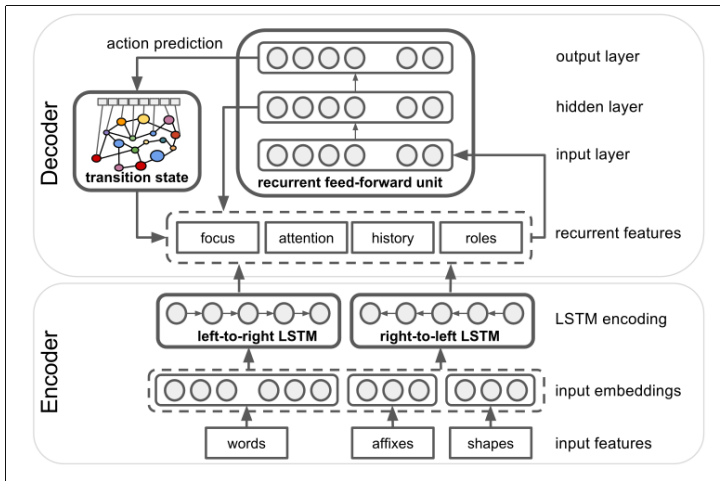
[95] Original: Seq2SQL takes as input a question and the columns of a table. It generates the corresponding SQL query, which, during training, is executed against a database. The result of the execution is utilized as the reward to train the reinforcement learning algorithm.

Papers and Tutorials **Nov 13, 2017**

SLING: A Framework for Frame Semantic Parsing

Table of Contents Local *Written by Brandon McKinzie*

M. Ringgaard, R. Gupta, F. Pereira, "SLING: A framework for frame semantic parsing" (2017)



Model.

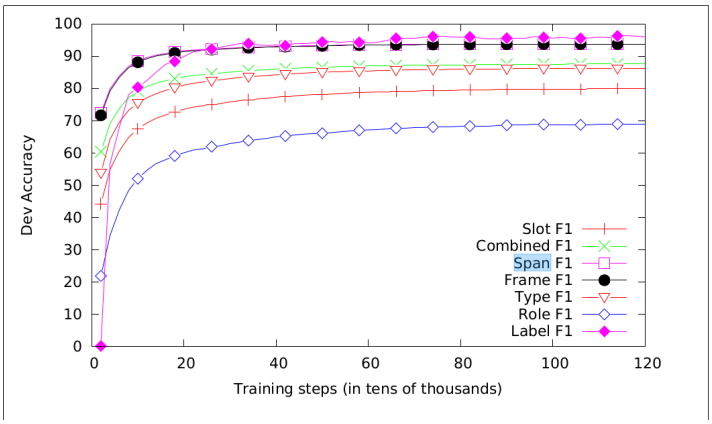
- **Inputs**. [words; affixes; shapes]
- **Encoder**.
  1. Embed.
  2. Bidirectional LSTM.
- **Inputs to TBRU**.
  - BLSTM [forward and backward] hidden state for the current token in the parser state.
  - **Focus**. Hidden layer activations corresponding to the transition steps that evoked/brought into *focus* the top-k frames in the attention buffer.
  - **Attention**. Recall that we maintain an **attention buffer**: an ordered list of frames, where the order represents closeness to center of attention. The attention portion of inputs for the TBRU looks at the top-k frames in the attention buffer, finds the phrases in the text (if any) that evoked them. The activations from the BLSTM for the last token of each of those phrases are included as TBRU inputs[96]
  - **History**. Hidden layer activations from the previous k steps.
  - **Roles**. Embeddings of (s_i, r_i, t_i), where the frame at position s_i in the attention buffer has a role (key) r_i with frame at position t_i as its value. Back-off features are added for the source roles (s_i, r_i), target role (r_i, t_i), and unlabeled roles (s_i, t_i).
- **Decoder** (TBRU). Outputs a softmax over possible transitions (actions).

[96] Okay, how is this attention at all? Seems misleading to call it attention.

Transition System. Below is the list of possible actions. Note that, since the system is trained to predict the correct *frame graph* result, it isn't directly told what order it should take a given set of actions[97].

- **SHIFT**. Move to next input token.
- **STOP**. Signal that we've reached end of parse.
- **EVOKE(type, num)**. New frame of **type** from next **num** tokens in the input; placed at front of attention buffer.
- **REFER(frame, num)**. New mention from next **num** tokens, evoking existing **frame** from attention buffer. Places at front.
- **CONNECT(source-frame, role, target-frame)**. Inserts **(role, target-frame)** slot into **source-frame**, move **source-frame** to front.
- **ASSIGN(source-frame, role, value)**. Same as CONNECT, but with primitive/constant **value**.
- **EMBED(target-frame, role, type)**. New frame of **type**, and inserts **(role, target-frame)** slot. New frame placed to front.
- **ELABORATE(source-frame, role, type)**. New frame of **type**. Inserts **(role, new-frame)** slot to **source-frame**. New frame placed at front.

Evaluation. Need some way of comparing an annotated document with its gold-standard annotation. This is done by constructing a virtual graph where the document is the start node. It is then connected to the spans (which are presumably nodes themselves), and the spans are connected to the frames they evoke. Frames that refer to other frames are given corresponding edges between them. Quality is computed by aligning the golden and predicted graphs and computing precision, recall, and F1. Specifically, these scores are computed separately for spans, frames, frame types, roles linking to other frames (referred to here as just "roles"), and roles that link to global constants (referred to here as just "labels"). Results are shown below.



[97]This is important to keep in mind, since more than one sequence of actions can result in a given predicted frame graph.

Poincaré Embeddings for Learning Hierarchical Representations

Written by Brandon McKinzie

M. Nickel and D. Kiela, "Poincaré Embeddings for Learning Hierarchical Representations" (2017)

Introduction. Dimensionality of embeddings can become prohibitively large when needed for complex data. Authors focus on mitigating this problem for large datasets whose objects can be organized according to a latent hierarchy[98]. They propose to compute embeddings in a particular model of hyperbolic space, the **Poincaré ball model**, claiming it is well-suited for gradient-based optimization (they make use of **Riemannian optimization**).

Prerequisite Math. Recall that a hyperbola is a set of points, such that for any point P of the set, the absolute difference of the distances $|PF_1|$, $|PF_2|$ to two fixed points F_1, F_2 (the foci), is constant, usually denoted by $2a$, $a > 0$. We can define a hyperbola by this set of points or by its canonical form, which are both given, respectively, as:

$$H = \{P| \; ||PF_2| - |PF_1|| = 2a\} \tag{315}$$

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{316}$$

where $b^2 := c^2 - a^2$, $(\pm a, 0)$ are the two vertices, and $(\pm c, 0)$ are the two foci. Cannon et al. define n-dimensional hyperbolic space by the formula

$$H^n = \{x \in \mathbb{R}^{n+1} : \; x * x = -1\} \tag{317}$$

where $*$ denotes the non-euclidean inner product (subtracts last term; same as Minkowski sapce-time). Notice that this is the defining equation for a hyperboloid of two sheets, and Cannon et al. says "usually we deal only with one of the two sheets." Hyperbolic spaces are well-suited to model hierarchical data, since both circle length and disc area grow *exponentially* with r.

[98] This begs the question: how useful would a Poincaré embedding be for situations where this assumption isn't valid?

Poincaré Embeddings. Let $\mathcal{B}^d = \{\boldsymbol{x} \in \mathbb{R}^d \mid ||\boldsymbol{x}|| < 1\}$ be the open d-dimensional unit ball. The **Poincaré ball** model of hyperbolic space corresponds then to the Riemannian manifold[99] $(\mathcal{B}^d, g_{\boldsymbol{x}})$, where

$$g_{\boldsymbol{x}} = \left(\frac{2}{1 - ||\boldsymbol{x}||^2}\right)^2 g^E \tag{318}$$

is the **Riemannian metric tensor**, and g^E denotes the Euclidean metric tensor. The distance between two points $\boldsymbol{u}, \boldsymbol{b} \in \mathcal{B}^d$ is given as

$$d(\boldsymbol{u}, \boldsymbol{v}) = \operatorname{arccosh}\left(1 + 2\frac{||\boldsymbol{u} - \boldsymbol{v}||^2}{(1 - ||\boldsymbol{u}||^2)(1 - ||\boldsymbol{v}||^2)}\right) \tag{319}$$

The boundary of the ball corresponds to the sphere S^{d-1} and is denoted by $\partial\mathcal{B}$. Geodesics in $\mathcal{B}^d$ are then circles that are orthogonal to $\partial\mathcal{B}$. To compute Poincaré embeddings for a set of symbols $\mathcal{S} = \{x_i\}_{i=1}^n$, we want to find embeddings $\Theta = \{\boldsymbol{\theta}_i\}_{i=1}^n$, where $\boldsymbol{\theta}_i \in \mathcal{B}^d$. Given some loss function $\mathcal{L}$ that encourages semantically similar objects to be close as defined by the Poincaré distance, our goal is to solve the optimization problem

$$\Theta' \leftarrow \arg\min_{\Theta} \mathcal{L}(\Theta) \qquad s.t.\ \forall \boldsymbol{\theta}_i \in \Theta : ||\boldsymbol{\theta}_i|| < 1 \tag{320}$$

Optimization. Let $\mathcal{T}_\theta\mathcal{B}$ denote the **tangent space** of a point $\boldsymbol{\theta} \in \mathcal{B}^d$. Let $\nabla_R \in \mathcal{T}_{\boldsymbol{\theta}}\mathcal{B}$ denote the **Riemannian gradient** of $\mathcal{L}(\boldsymbol{\theta})$, and ∇_E the Euclidean gradient of $\mathcal{L}(\boldsymbol{\theta})$. Using **RSGD**, parameter updates take the form

$$\boldsymbol{\theta}_{t+1} = \mathfrak{R}_{\boldsymbol{\theta}_t}\left(-\eta_t \nabla_R \mathcal{L}(\boldsymbol{\theta}_t)\right) \tag{321}$$

where $\mathfrak{R}_{\boldsymbol{\theta}_t}$ denotes the **retraction** onto $\mathcal{B}$ at $\boldsymbol{\theta}$ and η_t denotes the learning rate at time t.

[99] All five analytic models of hyperbolic geometry in Cannon et al. are differentiable manifolds with a Riemannian metric. A **Riemannian metric** $\mathrm{d}s^2$ on Euclidean space $\mathbb{R}^n$ is a function that assigns at each point $p \in \mathbb{R}^n$ a positive definite symmetric inner product on the tangent space at p, this inner product varying differentiably with the point p. If $x_1, \ldots, x_n$ are the standard coordinates in $\mathbb{R}^n$, then $\mathrm{d}s^2$ has the form $\sum_{i,j} g_{ij}\mathrm{d}x_i\mathrm{d}x_j$, and the matrix (g_{ij}) depends differentiably on x and is positive definite and symmetric.

Enriching Word Vectors with Subword Information (FastText)

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information" (2017)

Overview. Based on the skipgram model, but where each word is represented as a bag of character n-grams. A vector representation is associated *each* character n-gram; words being represented as the *sum*[100] of these representations.

Skipgram with Negative Sampling. Since this is based on skipgram, recall the objective of skipgram which is to maximize:

$$\sum_{t=1}^{T} \sum_{c \in \mathcal{C}_t} \log \Pr\left[w_c \mid w_t\right] \tag{322}$$

for a sequence of words $w_1, \dots w_T$. One way of parameterizing $\Pr\left[w_c \mid w_t\right]$ is by computing a softmax over a scoring function $s : (w_t, w_c) \mapsto \mathbb{R}$,

$$\Pr\left[w_c \mid w_t\right] = \frac{e^{s(w_t, w_c)}}{\sum_{j=1}^{W} e^{s(w_t, j)}} \tag{323}$$

However, this implies that, given w_t, we only predict one context word w_c. Instead, we can frame the problem as a set of independent binary classification tasks, and independently predict the presence/absence of context words. Let $\ell : x \mapsto \log(1 + e^{-x})$ denote the standard logistic negative log-likelihood. Our objective is:

$$\sum_{t=1}^{T} \left[\sum_{c \in \mathcal{C}_t} \ell(s(w_t, w_c)) + \sum_{n \in \mathcal{N}_{t,c}} \ell(-s(w_t, n)) \right] \tag{324}$$

where $\mathcal{N}_{t,c}$ is a set of negative examples sampled from the vocabulary. A common scoring function involves associating a distinct input vector u_w and output vector v_w for each word w. Then the score is computed as $s(w_t, w_c) = \boldsymbol{u}_{w_t}^T \boldsymbol{v}_{w_c}$.

[100] It would be interesting to explore other aggregation operations than just summation.

FastText. Main contribution is a different scoring function s that utilizes subword information. Each word w is represented as a bag of character n-grams. Special symbols $<$ and $>$ delimit word boundaries, and the authors also insert the special sequence containing the full word (with the delimiters) in its bag of n-grams. The word *where* is thus represented by first building its bag of n-grams, for the choice of $n = 3$:

$$\text{where} \longrightarrow \{< wh, \quad whe, \quad her, \quad ere, \quad re >, \quad < where >\} \tag{325}$$

Such a set of n-grams for a word w is denoted $\mathcal{G}_w$. Each n-gram g for a word w has its own vector $\boldsymbol{z}_g$, and the final vector representation of w is the sum of these. The scoring function becomes

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \boldsymbol{z}_g^T \boldsymbol{v}_c \tag{326}$$

DeepWalk: Online Learning of Social Representations

Written by Brandon McKinzie

B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," (2014).

Problem Definition. Classifying members of a social network into one or more categories.

> *Let $G = (V, E)$, where V are the members of the network, and E be its edges, $E \subseteq (V \times V)$. Given a partially labeled social network $G_L = (V, E, X, Y)$, with attributes $X \in \mathbb{R}^{|V| \times S}$ where S is the size of the feature space for each attribute vector, and $Y \in \mathbb{R}^{|V| \times |\mathcal{Y}|}$, $\mathcal{Y}$ is the set of labels.*

In other words, the elements of our training dataset, (X, Y), are the members of the social network, and we want to label each member, represented by a vector in $\mathbb{R}^S$, with one or more of the $|\mathcal{Y}|$ labels. We aim to learn features that capture the graph structure *independent* of the labels' distribution, and to do so in an unsupervised fashion.

Learning Social Representations. We want the representations to be adaptable, community-aware, low-dimensional, and continuous. The authors' method learns representations for vertices from a stream of short random walks, optimized with techniques from language modeling.

- **Random Walks**. Denote a random walk rooted at vertex v_i as $\mathcal{W}_{v_i}$, where the kth visited vertex is chosen at random from the neighbors of the $(k-1)^{th}$ visited vertex, and so on. Motivation for their use here is that they're "the foundation of a class of *output sensitive* algorithms which use them to compute local community structure information in time sublinear to the size of the input graph."
- **Language Modeling**. Authors present a generalization of language modeling, which traditionally aims to maximize $\Pr[w_n \mid w_0, \ldots, w_{n-1}]$ over all words in a training corpus. The motivation of the generalization is to explore the graph through a stream of short random walks. The walks are thought of as short sentences/phrases in a special language, and we want to estimate the probability of observing vertex v_i given all previous vertices so far in the random walk. Since we want to learn a latent social representation of each vertex, and not simply a probability distribution over node co-occurrences, we condition on the *embeddings* of visited nodes in this latent space (rather than the nodes themselves directly)

$$\Pr[v_i \mid \Phi(v_1), \Phi(v_2), \ldots, \Phi(v_{i-1})] \tag{327}$$

  where, in practice, the mapping function Φ is represented by a $|V| \times d$ matrix of free parameters (an embedding matrix). Since this becomes infeasible to compute as the walk length grows, the authors opt for an approach resembling CBOW: minimizing the NLL of vertices in the the context of a given vertex.

$$\min_{\Phi} \quad -\log \Pr[v_{i-w}, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{i+w} \mid \Phi(v_i)] \tag{328}$$

Remember that, here, v_j is the jth vertex visited in some given random walk.

DeepWalk Algorithm. Below is a conceptual summary of the procedure, followed by a figure/illustration of the formal algorithm definition.

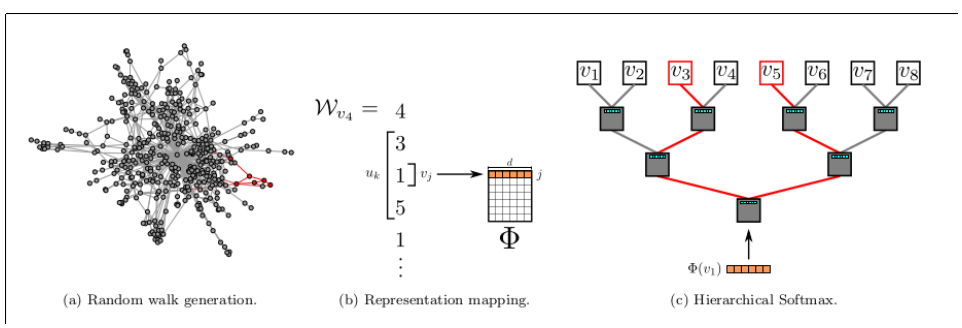
1. **Inputs**. Graph $G(V, E)$, window size w, embedding size d, walks per vertex γ, walk length t.
2. **Random Walk**. For each vertex v_i, compute $\mathcal{W}_{v_i} := RandomWalk(G, v_i, t)$.
3. **Updates**. Upon finishing a walk, $\mathcal{W}_{v_i}$, run skipgram on the sequence of walked vertices to update the embedding matrix Φ.
4. **Outputs**. The embedding matrix $\Phi \in \mathbb{R}^{|V| \times d}$.

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

1: Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
2: Build a binary Tree T from V
3: **for** $i = 0$ to γ **do**
4: $\quad \mathcal{O} = \text{Shuffle}(V)$
5: $\quad$ **for each** $v_i \in \mathcal{O}$ **do**
6: $\quad\quad \mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
7: $\quad\quad$ SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
8: $\quad$ **end for**
9: **end for**

where $SkipGram(\Phi, W_{v_i}, w)$ performs SGD updates on Φ to minimize $-\log \Pr[u_k \mid \Phi(v_j)]$ for each visited v_j, for each u_k in the "context" of v_j. Notice that a binary tree T is build from the set of vertices V (line 2) – this is done as preparation for computing each $\Pr[u_k \mid \Phi(v_j)]$ via a *hierarchical softmax*, to reduce computational burden of its partition function. Finally, a visual overview of the DeepWalk algorithm is shown below. The authors use this algorithm, com-



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

bined with a one-vs-rest logistic regression implementation by LibLinear, for various multiclass multilabel classification tasks.

Review of Relational Machine Learning for Knowledge Graphs

Written by Brandon McKinzie

Nickel, Murphy, Tresp, and Gabrilovich, "Review of Relational Machine Learning for Knowledge Graphs," (2015).

Introduction. Paper discusses latent feature models such as tensor factorization and multiway neural networks, and mining observable patterns in the graph. In **Statistical Relational Learning (SRL)**, the representation of an object can contain its relationships to other objects. The main goals of SRL include:

- Prediction of missing edges (relationships between entities).
- Prediction of properties of nodes.
- Clustering nodes based on their connectivity patterns.

We'll be reviewing how SRL techniques can be applied to large-scale **knowledge graphs** (KGs), i.e. graph structured knowledge bases (KBs) that store factual information in the form of relationships between entities.

Probabilistic Knowledge Graphs. Let $\mathcal{E} = \{e_1, \ldots, e_{N_e}\}$ be the set of all entities and $\mathcal{R} = \{r_1, \ldots, r_{N_r}\}$ be the set of all relation types in a KG. We model each *possible* triple $x_{ijk} = (e_i, r_k, e_j)$ as a binary random variable $y_{ijk} \in \{0, 1\}$ that indicates its existence. The full tensor $\boldsymbol{Y} \in \{0,1\}^{N_e \times N_e \times N_r}$ is called the *adjacency tensor*, where each possible realization of $\boldsymbol{Y}$ can be interpreted as a possible world.

Clearly, $\boldsymbol{Y}$ will be large and sparse in most applications. Ideally, a relational model for large-scale KGs should scale at most linearly with the data size, i.e., linearly in the number of entities N_e, linearly in the number of relations N_r, and linearly in the number of *observed* triples $|\mathcal{D}| = N_d$.

Types of SRL Modesls. The presence or absence of certain triples in relational data is correlated with (i.e. predictive of) the presence or absence of certain other triples. In other words, the random variables y_{ijk} are correlated with each other. There are three main ways to model these correlations:

1. **Latent feature models**: Assume all y_{ijk} are conditionally independent given latent features associated with the subject, object and relation type and additional parameters.
2. **Graph feature models**: Assume all y_{ijk} are conditionally independent given observed graph features and additional parameters.
3. **Markov Random Fields**: Assume all y_{ijk} have local interactions.

The first two model classes predict the existence of a triple x_{ijk} via a score function $f(x_{ijk}; \Theta)$

which represents the model's confidence that a triple exists given the parameters Θ. The conditional independence assumptions can be written as

$$\Pr\left[\boldsymbol{Y} \mid \mathcal{D}, \Theta\right] = \prod_{i=1}^{N_e} \prod_{j=1}^{N_e} \prod_{k=1}^{N_r} \operatorname{Ber}\left(y_{ijk} \mid \sigma\left(f(x_{ijk}; \Theta)\right)\right) \tag{329}$$

where Ber is the Bernoulli distribution[101]. Such models will be referred to as *probabilistic models*. We will also discuss *score-based models*, which optimize $f(\cdot)$ via maximizing the margin between existing and non-existing triples.

Latent Feature Models. We assume the variables y_{ijk} are conditionally independent given a set of global latent features and parameters. All LFMs explain triples (observable facts) via latent features of entities[102]. One task of all LFMs is to infer these [latent] features automatically from the data.

- **RESCAL**: a bilinear model. Models the score of a triple x_{ijk} as

$$f_{ijk}^{RESCAL} := \boldsymbol{e}_i^T \boldsymbol{W}_k \boldsymbol{e}_j = \sum_{a=1}^{H_e} \sum_{b=1}^{H_e} w_{abk} e_{ia} e_{jb} \tag{331}$$

  where the entity vectors $\boldsymbol{e}_i \in \mathbb{R}^{H_e}$ and H_e denotes the number of latent features in the model. The parameters of the model are $\Theta = \{\{\boldsymbol{e}_i\}_{i=1}^{N_e}, \{\boldsymbol{W}_k\}_{k=1}^{N_r}\}$. Note that entities have the same latent representation regardless of whether they occur as subjects or objects in a relationship (shared representation), *thus allowing the model to capture global dependencies in the data*. We can make a connection to **tensor factorization** methods by seeing that the equation above can be written compactly as

$$\boldsymbol{F}_k = \boldsymbol{E} \boldsymbol{W}_k \boldsymbol{E}^T \tag{332}$$

  where $\boldsymbol{F}_k \in \mathbb{R}^{N_e \times N_e}$ is the matrix holding all scores for the k-th relation, and the ith row of $\boldsymbol{E} \in \mathbb{R}^{N_e \times H_e}$ holds the latent representation of $\boldsymbol{e}_i$.

- **Multi-layer perceptrons**. We can rewrite RESCAL as

$$f_{ijk}^{RESCAL} := \boldsymbol{w}_k^T \boldsymbol{\phi}_{i,j}^{RESCAL} \tag{333}$$

$$\boldsymbol{\phi}_{i,j}^{RESCAL} := \boldsymbol{e}_j \otimes \boldsymbol{e}_i \tag{334}$$

  where $\boldsymbol{w}_k = \operatorname{vec}(\boldsymbol{W}_k)$ (vector of size H_e^2 obtained by stacking columns of $\boldsymbol{W}_k$). The

[101] Notation used:

$$\operatorname{Ber}(y \mid p) = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases} \tag{330}$$

[102] It appears that "latent" is being used here synonymously with "not directly observed in the data".

authors extend this to what they call the E-MLP (E for entity) model:

$$f_{ijk}^{E-MLP} := \boldsymbol{w}_k^T \boldsymbol{g}(\boldsymbol{h}_{ijk}^a) \tag{335}$$

$$\boldsymbol{h}_{ijk}^a := \boldsymbol{A}_k^T \boldsymbol{\phi}_{ij}^{E-MLP} \tag{336}$$

$$\boldsymbol{\phi}_{ij}^{E-MLP} := [\boldsymbol{e}_i; \boldsymbol{e}_j] \tag{337}$$

Graph Feature Models. Here we assume that the existence of an edge can be predicted by extracting features from the observed edges in the graph. In contrast to LFMs, this kind of reasoning explains triples directly from the observed triples in the KG.

- **Similarity measures for uni-relational data**. Link prediction in graphs that consist only of a single relation (e.g. (Bob, isFriendOf, Sally) for a social network). Various **similarity indices** have been proposed to measure similarity of entities, of which there are three main classes:
  1. **Local** similarity indices: Common Neighbors, Adamic-Adar index, Preferential Attachment derive entity similarities from number of common neighbors.
  2. **Global** similarity indices: Katz index, Leicht-Holme-Newman index (ensembles of all paths bw entities). Hitting Time, Commute Time, PageRank (random walks).
  3. **Quasi-local** similarity indices: Local Katz, Local Random Walks.
- **Path Ranking Algorithm** (**PRA**): extends the idea of using random walks of bounded lengths for predicting links in multi-relational KGs. Let $\pi_L(i, j, k, t)$ denote a path of length L of the form $e_i \xrightarrow{r_1} e_2 \xrightarrow{r_2} e_3 \cdots \xrightarrow{r_L} e_j$, where t represents the sequence of edge types $t = (r_1, r_2, \ldots, r_L)$. We also require there to be a direct arc $e_i \xrightarrow{r_k} e_j$, representing the existence of a relationship of type k from e_i to e_j. Let $\Pi_L(i, j, k)$ represent the <u>set</u> of all such paths of length L, ranging over path types t.

  We can compute the probability of following a given path by assuming that at each step we follow an outgoing link uniformly at random. Let $\Pr[\pi_L(i, j, k, t)]$ be the probability of the path with type t. *The key idea in PRA is to use these path probabilities as features for predicting the probabilities of missing edges.* More precisely, the feature vector and score function (logistic regression) are as follows:

$$\boldsymbol{\phi}_{ijk}^{PRA} = [\Pr[\pi] : \pi \in \Pi_L(i, j, k)] \tag{338}$$

$$f_{ijk}^{PRA} := \boldsymbol{w}_k^T \boldsymbol{\phi}_{ijk}^{PRA} \tag{339}$$

TODO: Finish...

Papers and Tutorials **Dec 6, 2017**

Fast Top-K Search in Knowledge Graphs

Table of Contents Local *Written by Brandon McKinzie*

S. Yang, F. Han, Y. Wu, X. Yan, "Fast Top-K Search in Knowledge Graphs."

Introduction. Task: Given a knowledge graph G, scoring function F, and a graph query Q, top-k subgraph search over G returns k answers with the highest matching scores. An example is searching for movie makers (directors) worked with "Brad" and have won awards, illustrated below:

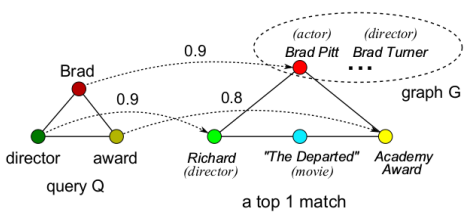


Fig. 1: Top-k subgraph querying

Clearly, it would be extremely inefficient to enumerate all possible matches and then rank them.

Preliminaries/Terminology.

- **Queries**. A query [graph] is defined as $Q = (V_Q, E_Q)$. Each query node in Q provides information/constraints about an entity, and an edge between two nodes specifies the relationship or the connectivity constraint posed on the two nodes. $Q*$ denotes a star-shaped query, which is basically a graph that looks like a star (central node with tree-like structure radially outward).
- **Subgraph Matching**. Given a graph query Q and a knowledge graph G, a <u>match</u> $\phi(Q)$ of Q in G is a subgraph of G, specified by a one-to-one matching function ϕ. It maps each node u (edge $e = (u, u')$) in Q to a node match $\phi(u)$ (edge match $\phi(e) = (\phi(u), \phi(u'))$) in $\phi(Q)$.

The matching score between query Q and its match $\phi(Q)$ is

$$F(\phi(Q)) = \sum_{v \in V_Q} F_V(v, \phi(v)) + \sum_{e \in E_Q} F_E(e, \phi(e)) \tag{340}$$

$$F_V(v, \phi(v)) = \sum_i \alpha_i f_i(v, \phi(v)) \tag{341}$$

$$F_E(e, \phi(e)) = \sum_j \beta_j f_j(e, \phi(e)) \tag{342}$$

Star-Based Top-K Matching.

1. **Query decomposition**: Given query Q, STAR decomposes Q to a set of star queries $\mathcal{Q}$. A star query contains a pivot node and a set of leaves as its neighbors in Q.
2. **Star querying engine**: Generate a set of top matches for each star query $\mathcal{Q}$.
3. **Top-k rank join**. The top matches for multiple star queries are collected and joined to produce top-k complete matches of Q.

Dynamic Recurrent Acyclic Graphical Neural Networks (DRAGNN)

Written by Brandon McKinzie

Kong et al., "DRAGNN: A Transition-based Framework for Dynamically Connected Neural Networks," (2017).

Transition Systems. Define a transition system $\mathcal{T} \triangleq \{\mathcal{S}, \mathcal{A}, t\}$, where

- $\mathcal{S} = \mathcal{S}(x)$ is a set of states, where that set depends on the input sequence x.
- A special start state $s^\dagger \in \mathcal{S}(x)$.
- A set of allowed decisions $\mathcal{A}(s, x)$ $\forall s \in \mathcal{S}(x)$.
- A transition function $t(s, d, x)$ returning a new state s' for any decision $d \in \mathcal{A}(s, x)$.

The authors then define a **complete structure** as a sequence of state/decision pairs $(s_1, d_1) \dots (s_n, d_n)$ such that $s_1 = s^\dagger$, $d_i \in \mathcal{A}(s_i)$ for $i = 1, \dots, n$ and $s_{i+1} = t(s_i, d_i)$, where $n = n(x)$ is the number of decisions for input x[103]. We'll use transition systems to map inputs x into a sequence of output symbols $d_1, \dots, d_n$.

Transition Based Recurrent Networks. When combining transition systems with recurrent networks, we will refer to them as **Transition Based Recurrent Units (TBRU)**, which consist of:

- Transition system $\mathcal{T}$.
- Input function $\boldsymbol{m}(s) : \mathcal{S} \mapsto \mathbb{R}^K$ that maps states to some fixed-size vector representation (e.g. an embedding lookup operation).
- Recurrence function $\boldsymbol{r}(s) : \mathcal{S} \mapsto \mathbb{P}\{1, \dots, i-1\}$ that maps states to a set of previous time steps, where $\mathbb{P}$ is the power set. Note that $|\boldsymbol{r}(s)|$ may vary with s. We use $\boldsymbol{r}$ to specify state-dependent recurrent links in the unrolled computation graph.
- The RNN cell $\boldsymbol{h}_s \leftarrow \mathbf{RNN}(\boldsymbol{m}(s), \{\boldsymbol{h}_i \mid i \in \boldsymbol{r}(s)\})$.

Example: Sequential tagging RNN. Let $\boldsymbol{x} = \{x_1, \dots, x_n\}$ be a sequence of input tokens. Let the ith output, d_i, be a tag from some predefined set of tags $\mathcal{A}$. Then our model can be defined as:

- Transition system: $\mathcal{T} = \{\ s_i{=}\mathcal{S}(x_i){=}\{1, \dots, d_{i-1}\},\ \ \mathcal{A},\ \ t(s_i, d_i, x_i){=}s_{i+1}{=}\ s_i{+}\{d_i\}\ \}$.
- Input function: $\boldsymbol{m}(s_i) = embed(x_i)$.
- Recurrence function: $\boldsymbol{r}(s_i) = \{i-1\}$ to connect the network to the previous state.
- RNN cell: $\boldsymbol{h}_i \leftarrow LSTM(\boldsymbol{m}(s_i) \mid \boldsymbol{r}(s_i) = \{i-1\})$.

[103] The authors state that we are only concerned with complete structures that have the same number of decisions $n(x)$ for the same input x.

Example: Parsey McParseface.

- Transition system: the **arc-standard** transition system, defined in image below[104].

Initialization:	$c_s(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset)$	
Terminal:	$C_t = \{c \in C \mid c = ([0], [\,], A)\}$	
Transitions:	$(\sigma, [i\|\beta], A) \Rightarrow ([\sigma\|i], \beta, A)$	(SHIFT)
	$([\sigma\|i\|j], B, A) \Rightarrow ([\sigma\|j], B, A \cup \{(j, l, i)\})^1$	(LEFT-ARC$_l$)
	$([\sigma\|i\|j], B, A) \Rightarrow ([\sigma\|i], B, A \cup \{(i, l, j)\})$	(RIGHT-ARC$_l$)

[1] Permitted only if $i \neq 0$.

FIGURE 1. The arc-standard stack-based transition system for projective dependency parsing. The notation $[\sigma|i]$ (for the stack) denotes a right-headed list with head $i$and tail σ; the notation $[j|\beta]$ (for the buffer) denotes a left-headed list with head j and tail β.

so the state contains all words and partially built trees (stack) as well as unseen words (buffer).
- Input function: $\boldsymbol{m}(s_i)$ is the concatenation of 52 feature embeddings extracted from tokens based on their positions in the stack and the buffer.
- Recurrence function: $\boldsymbol{r}(s_i)$ is empty, as this is a feed-forward network.
- RNN cell: a feed-forward MLP (so not an RNN...).

Inference with TBRUs. To predict the output sequence $\{d_1, \ldots, d_n\}$ given input sequence $\boldsymbol{x} = \{x_1, \ldots, x_n\}$, do:

1. Initialize $s_1 = s^\dagger$.
2. For $i = 1, \ldots, n$:
   (a) Compute $\boldsymbol{h}_i = \mathbf{RNN}(\boldsymbol{m}(s_i), \{\boldsymbol{h}_j \mid j \in \boldsymbol{r}(s_i)\})$.
   (b) Update transition state:

$$d_i \leftarrow \arg\max_{d \in \mathcal{A}(s_i)} \boldsymbol{w}_d^T \boldsymbol{h}_i \tag{343}$$

$$s_{i+1} \leftarrow t(s_i, d_i) \tag{344}$$

NOTE: This defines a locally normalized training procedure, whereas Andor et al. of Syntaxnet clearly conclude that their globally normalized model is the preferred choice.

[104] Image taken from "Transition-Based Parsing" by Joakim Nivre. Note that "right-headed" means "goes from left to right" or "headed *to* the right".

Combining multiple TBRUs. We connect multiple TBRUs with different transition systems via $\boldsymbol{r}(s)$.

1. We execute a list of T TBRU components sequentially, so that each TBRU advances a global step counter.
2. *Each* transition state, s^τ, from the τ'th component has access the *terminal* states from every prior transition system, and the recurrence function $\boldsymbol{r}(s^\tau)$ for any given component can pull hidden activations from every prior one as well.

Example: Multi-task bi-directional tagging. Say we want to do both POS and NER tagging (indices start at 1).

- Left-to-right: $\mathcal{T}$ = *shift-only*, $\boldsymbol{m}(s_i) = \boldsymbol{x}_i$, $\boldsymbol{r}(s_i) = \{i-1\}$.
- Right-to-left: $\mathcal{T}$ = *shift-only*, $\boldsymbol{m}(s_{n+i}) = \boldsymbol{x}_{(n-i)+1}$, $\boldsymbol{r}(s_{n+i}) = \{n+i-1\}$.
- POS Tagger: $\mathcal{T}_{POS}$ = *tagger*, $\boldsymbol{m}(s_{2n+i}) = \{\}$, $\boldsymbol{r}(s_{2n+i}) = \{i, (2n-i)+1\}$
- NER Tagger: $\mathcal{T}_{NER}$ = *tagger*, $\boldsymbol{m}(s_{3n+i}) = \{\}$, $\boldsymbol{r}(s_{3n+i}) = \{i, (2n-i)+1, 2n+i\}$

which illustrates the most important aspect of the TBRU:

> *A TBRU can serve as both an encoder for downstream tasks and a decoder for its own task simultaneously.*

For this example, the POS Tagger served as both an encoder for the NER task as well as a decoder for the POS task.

Training a DRAGNN. Assume training data consists of examples $\boldsymbol{x}$ along with gold decision sequences for a given TBRU in the DRAGNN. Given decisions $d_1, \ldots, d_N$ from prior components $1, \ldots, T-1$, the log-likelihood for training the T'th TBRU along its gold decision sequence $d^\star_{N+1}, \ldots, d^\star_{N+n}$ is then:

$$L(\boldsymbol{x},\ d^\star_{N+1:N+n};\ \theta) = \sum_i \log \Pr\left[d^\star_{N+i} \mid d_{1:N},\ d^\star_{N+1:N+i-1};\ \theta\right] \tag{345}$$

During training, the entire input sequence is unrolled and **backpropagation through structure** is used for gradient computation.

More Detail: Arc-Standard Transition System

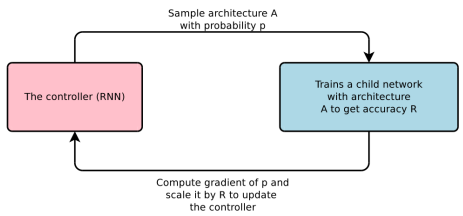
The arc-standard transition system is mentioned a lot, but with little detail. Here I'll synthesize what I find from external resources. The literature defines the states in a transition system slightly differently than the DRAGNN paper. Here we'll define them as a **configuration** $c = (\Sigma, B, A)$ triplet, where

- Σ is the **stack** of tokens in x that we've [partially] processed.
- B is the **buffer** of remaining tokens in x that we need to process.
- A is a set of **arcs** (w_i, w_j, ℓ) that link w_i to w_j, and label the arc/link as ℓ.

So, in the arc-standard transition system figure presented with Parsey McParseface earlier,

- SHIFT just means "move the head element of the buffer to the tail element of the buffer".
- Left-arc just means "make a link *from* the tail element of the stack *to* the element before it. Remove the pointed-to element from the stack."
- Right-arc just means "make a link *from* the element before the tail element in the stack *to* the tail element. Remove the pointed-to element from the stack."

B. Zoph and Q. Le, "Neural Architecture Search with Reinforcement Learning," (2017).



Controller RNN. Generates architectures with a predefined number of layers, which is increased manually as training progresses. At convergence, validation accuracy of the generated network is recorded. Then, the controller parameters θ_c are optimized to maximized the expected validation accuracy over a batch of generated architectures.

Reinforcement Learning to learn the controller parameters θ_c. Let $a_{1:T}$ denote a list of actions taken by the controller[105], which defines a generated architecture. We denote the resulting validation accuracy by R, which is the reward signal for our RL task. Concretely, we want our controller to maximize its expected reward, $J(\theta_c)$:

$$J(\theta_c) = \mathbb{E}_{P(a_{1:T};\ \theta_c)}\left[R\right] \tag{346}$$

Since the quantity $\nabla_{\theta_c} R$ is non-differentiable[106], we use a **policy gradient** method to iteratively update θ_c. All this means is that we instead compute gradients over the softmax outputs (the action probabilities), and use the value of R as a simple weight factor.

$$\nabla_{\theta_c} J(\theta_c) = R \sum_{t=1}^{T} \mathbb{E}_{P(a_{1:T};\ \theta_c)}\left[\nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c)\right] \tag{347}$$

$$\approx \frac{1}{m} \sum_{k=1}^{m} R_k \sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c) \tag{348}$$

where the second equation is the empirical approximation (batch-average instead of expectation) over a batch of size m, an unbiased estimator for our gradient[107]. Also note that we *do*

[105] Note that T is not necessarily the number of layers, since a single generated layer can correspond to multiple actions (e.g. stride height, stride width, num filters, etc.).

[106] R is a function of the action sequence $a_{1:T}$ and the parameters θ_c, and implicitly depends on the samples used for the validation set. Clearly, we do not have access to an analytical form of R, and computing numerical gradients via small perturbations of $theta_c$ is computationally intractable.

[107] It is unbiased for the same reason that any average over samples x drawn from a distribution $P(x)$ is an unbiased estimator for $\mathbb{E}_P\left[x\right]$.

have access to the distribution $P(a_{1:T}; \theta_c)$ since it is *defined* to be the joint softmax probabilities of our controller, given its parameter values θ_c (i.e. this is not a p_{data} vs p_{model} situation). The approximation 347 is an unbiased estimator for 348, but has high variance. To reduce the variance of our estimator, the authors employ a baseline function b that does not depend on the current action:

$$\frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c)(R_k - b) \tag{349}$$

Joint Extraction of Events and Entities within a Document Context

Written by Brandon McKinzie

B. Yang and T. Mitchell, "Joint Extraction of Events and Entities within a Document Context," (2016).

Introduction. Two main reasons that state-of-the-art event extraction systems have difficulties:

1. They extract events and entities in separate stages.
2. They extract events independently from each individual sentence, ignoring the rest of the document.

This paper proposes an approach that simultaneously extracts events and entities within a document context. They do this by first decomposing the problem into 3 tractable subproblems:

1. Learning the dependencies between a single event [trigger] and all of its potential arguments.
2. Learning the co-occurrence relations between events across the document.
3. Learning for entity extraction.

and then combine these learned models into a joint optimization framework.

Learning Within-Event Structures. For now, assume we have some document x, a set of candidate event triggers $\mathcal{T}$, and a set of candidate entities $\mathcal{N}$. Denote the set of entity candidates that are potential arguments for trigger candidate i as $\mathcal{N}_i$. The joint distribution over the possible trigger types, roles, and entities for those roles, is given by

$$\Pr_{\boldsymbol{\theta}}\left[t_i, \boldsymbol{r}_i, \boldsymbol{a} \mid i, \mathcal{N}_i, x\right] \propto \tag{350}$$

$$\exp\left(\boldsymbol{\theta}_1^T f_1(t_i) + \sum_{j \in \mathcal{N}_i}\left[\boldsymbol{\theta}_2^T f_2(r_{ij}) + \boldsymbol{\theta}_3^T f_3(t_i, r_{ij}) + \boldsymbol{\theta}_4^T f_4(a_j) + \boldsymbol{\theta}_5^T f_5(r_{ij}, a_j)\right]\right) \tag{351}$$

All f_i also depend on i, x. In addition, all f_i except f_1 depend on the current j in the summation.

where each f_i is a feature function, and I've colored the unary feature functions green. The unary features are tabulated in Table 1 of the original paper. They use simple indicator functions $1_{t,r}$ and $1_{r,a}$ for the pairwise features. They train using maximum-likelihood estimates with L2 regularization:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) - \lambda ||\boldsymbol{\theta}||_2^2 \tag{352}$$

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_i \log\left(\Pr_{\boldsymbol{\theta}}\left[t_i, \boldsymbol{r}_i, \boldsymbol{a} \mid i, \mathcal{N}_i, x\right]\right) \tag{353}$$

and use **L-BFGS** to optimize the training objective.

Learning Event-Event Relations. A pairwise model of event-event relations in a document. Training data consists of all pair of trigger candidates that co-occur in the same sentence or are connected by a co-referent subject/object if they're in different sentences. Given a trigger candidate pair (i, i'), we estimate the probabilities for their event types $(t_i, t_{i'})$ as

$$\Pr_\phi \left[t_i, t_{i'} \mid x, i, i'\right] \propto \exp\left(\phi^T g(t_i, t_{i'}, x, i, i')\right) \tag{354}$$

where g is a feature function that depends on the trigger candidate pair and their context. In addition to re-using the trigger features in Table 1 of the paper, they also introduce relational trigger features:

1. whether they're connected by a conjunction dependency relation
2. whether they share a subject or an object
3. whether they have the same head word lemma
4. whether they share a semantic frame based on FrameNet.

As before, they using L-BFGS to compute the maximum-likelihood estimates of the parameters ϕ.

Entity Extraction. Trained a standard linear-chain **CRF** using the BIO scheme. Their CRF features:

1. current words and POS tags
2. context words in a window of size 2
3. word type such as all-capitalized, is-capitalized, all-digits
4. Gazetteer-based entity type if the current word matches an entry in the gazetteers collected from Wikipedia.
5. pre-trained word2vec embeddings for each word

Joint Inference. Allows information flow among the 3 local models and finds globally-optimal assignments of all variables. Define the following objective:

$$\max_{\boldsymbol{t}, \boldsymbol{r}, \boldsymbol{a}} \sum_{i \in \mathcal{T}} E(t_i, \boldsymbol{r}_i, \boldsymbol{a}) + \sum_{i, i' \in \mathcal{T}} R(t_i, t_{i'}) + \sum_{j \in \mathcal{N}} D(a_j) \tag{355}$$

where

- The first term is the sum of confidence scores for individual event mentions from the within-event model.

$$E(t_i, \boldsymbol{r}_i, \boldsymbol{a}) = \log p_\theta(t_i) + \sum_{j \in \mathcal{N}_i} \left[\log p_\theta(t_i, r_{ij}) + \log p_\theta(r_{ij}, a_j)\right] \tag{356}$$

- The second term is the sum of confidence scores for event relations based on the pairwise event model.

$$R(t_i, t_{i'}) = \log p_\phi(t_i, t_{i'} \mid i, i', x) \tag{357}$$

- The third term is sum of confidence scores for entity mentions, where

$$D(a_j) = \log p_\psi(a_j \mid j, x) \tag{358}$$

  and $p_\psi(a_j \mid j, x)$ is the marginal probability derived from the linear-chain CRF.

The optimization is subject to agreement constraints that enforce the overlapping variables among the 3 components to agree on their values. The joint inference problem can be formulated as an **integer linear problem (ILP)**[108]. To solve it efficiently, they find solutions for the relaxation of the problem using a **dual decomposition algorithm** AD^3.

[108] From Wikipedia: An integer linear program in canonical form: maximize $c^T x$ subject to $Ax \leq b, x \geq 0, x \in \mathbb{Z}^n$

Globally Normalized Transition-Based Neural Networks

D. Andor et al., "Globally Normalized Transition-Based Neural Networks," (2016).

Introduction. Authors demonstrate that simple FF neural networks can achieve comparable or better accuracies than LSTMs, as long as they are **globally normalized**. They don't use any recurrence, but perform **beam search** for maintaining multiple hypotheses and introduce global normalization with a **CRF** objective to overcome the label bias problem that locally normalized models suffer from.

Transition System. Given an input sequence x, define:

- Set of states $S(x)$.
- Start state $s^\dagger \in S(x)$.
- Set of decisions $\mathcal{A}(s, x)$ for all $s \in S(x)$.
- Transition function $t(s, d, x)$ returning new state s' for any decision $d \in A(s, x)$.

The scoring function $\rho(s, d; \theta)$, which gives the score for decision d in state s, will be defined:

$$\rho(s, d; \theta) = \phi(s; \theta^{(l)}) \cdot \theta^{(d)} \tag{359}$$

which is just the familiar logits computation for decision d. $\theta^{(l)}$ are the parameters of the network excluding the parameters at the final layer, $\theta^{(d)}$. $\phi(s; \theta^{(l)})$ gives the representation for state s computed by the neural network under parameters $\theta^{(l)}$.

Global vs. Local Normalization.

- **Local**. Conditional probabilities $\Pr[d_j \mid s_j; \theta]$ are normalized locally over the scores for each possible action d_j from the current state s_j.

$$\Pr_L[d_{1:n}] = \prod_{j=1}^{n} \Pr[d_j \mid s_j; \theta] = \frac{\exp\left(\sum_{j=1}^{n} \rho(s_j, d_j; \theta)\right)}{\prod_{j=1}^{n} Z_L(s_j; \theta)} \tag{360}$$

$$Z_L(s_j; \theta) = \sum_{d' \in \mathcal{A}(s_j)} \exp\left(\rho(s_j, d'; \theta)\right) \tag{361}$$

  Beam search can be used to attempt to find the action sequence with highest probability.
- **Global**. In contrast, a CRF defines:

$$\Pr_G[d_{1:n}] = \frac{\exp\left(\sum_{j=1}^{n} \rho(s_j, d_j; \theta)\right)}{Z_G(\theta)} \tag{362}$$

$$Z_G(\theta) = \sum_{d'_{1:n} \in \mathcal{D}_n} \exp\left(\sum_{j=1}^{n} \rho(s'_j, d'_j; \theta)\right) \tag{363}$$

where $\mathcal{D}_n$ is the set of all valid sequences of decisions of length n. The inference problem is now to find

$$\arg\max_{d_{1:n} \in \mathcal{D}_n} \Pr_G [d_{1:n}] = \arg\max_{d_{1:n} \in \mathcal{D}_n} \sum_{j=1}^{n} \rho(s_j, d_j; \theta) \tag{364}$$

and we can also use beam search to approximately find the argmax.

Training. SGD on the NLL of the data under the model. The NLL takes a different form depending on whether we choose a locally normalized model vs a globally normalized model.

- **Local**.

$$L_{local}(d^*_{1:n}; \theta) = -\ln \Pr_L [d^*_{1:n}; \theta] \tag{365}$$

$$= -\sum_{j=1}^{n} \rho(s^*_j, d^*_j; \theta) + \sum_{j=1}^{n} \ln Z_L(s^*_j; \theta) \tag{366}$$

- **Global**.

$$L_{global}(d^*_{1:n}; \theta) = -\ln \Pr_G [d^*_{1:n}; \theta] \tag{367}$$

$$= -\sum_{j=1}^{n} \rho(s^*_j, d^*_j; \theta) + \ln Z_G(\theta) \tag{368}$$

To make learning tractable for the globally normalized model, the authors use **beam search with early updates**, defined as follows. Keep track of the location of the gold path[109] in the beam as the prediction sequence is being constructed. If the gold path is not found in the beam after step j, run one step of SGD on the following objective:

$$L_{global-beam}(d^*_{1:j}, \theta) = -\sum_{t=1}^{j} \rho(d^*_{1:t-1}, d^*_t; \theta) - \ln \sum_{d'_{1:j} \in \mathcal{B}_j} \exp\left(\sum_{t=1}^{j} \rho(d'_{1:t-1}, d'_t; \theta)\right) \tag{369}$$

where $\mathcal{B}_j$ contains all paths in the beam at step j, *and* the gold path prefix $d^*1:j$. If the gold path remains in the beam throughout decoding, a gradient step is performed using $\mathcal{B}_T$, the beam at the end of decoding. When training the global model, the authors first pretrain[110] using the local objective function, and then perform additional training steps using the global objective function..

[109] The gold path is the predicted sequence that matches the true labeled sequence, up to the current timestep.

The Label Bias Problem. Locally normalized models often have a very weak ability to revise earlier decisions. Here we will prove that **globally normalized models are strictly more expressive than locally normalized models**[111]. Let $\mathcal{P}_L$ denote the set of all possible distributions $p_L(d_{1:n} \mid x_{1:n})$ under the local model as the scores ρ vary. Let $\mathcal{P}_G$ be the same, but for the global model.

We are assuming that both P_L and P_G consist of log-linear distributions of scoring functions $\rho(d_{1:t-1}, d_t, x_{1:t})$

Theorem 3.1 *$\mathcal{P}_L$ is a strict subset*[112] *of $\mathcal{P}_G$, that is $\mathcal{P}_L \subsetneq \mathcal{P}_G$.*

In other words, a globally normalized model can model any distribution that a locally normalized one can, but the converse is not true. I've worked through the proof below.

Proof: $P_L \subsetneq P_G$

Proof that $P_L \subseteq P_G$. For any locally normalized model with scores $\rho_L(d_{1:t-1}, d_t, x_{1:t})$, we can define a corresponding p_G over scores

$$\rho_G(d_{1:t-1}, d_t, x_{1:t}) = \ln p_L(d_t \mid d_{1:t-1}, x_{1:t}) \tag{370}$$

By definition, this means that $p_G(d_{1:t} \mid x_{1:t}) = p_L(d_{1:t} \mid x_{1:t})$.
Proof that $P_G \nsubseteq P_L$. A proof by example. Consider a dataset consisting entirely of one of the following tagged sequences:

$$\boldsymbol{x} = abc, \qquad \boldsymbol{d} = ABC \tag{371}$$
$$\boldsymbol{x} = abe, \qquad \boldsymbol{d} = ADE \tag{372}$$

Similar to a typical linear-chain CRF, let $\mathcal{T}$ denote the set of observed label transitions, and let $\mathcal{E}$ denote the set of observed (x_t, d_t) pairs. Let α be the single scalar parameter of this simple model, where

$$\rho(d_{1:t-1}, d_t, x_{1:t}) = \alpha \left(\mathbb{1}_{(d_{t-1}, d_t) \in \mathcal{T}} + \mathbb{1}_{(x_t, d_t) \in \mathcal{E}} \right) \tag{373}$$

for all t. This results in the following distributions p_G and p_L, evaluating on input sequence of length 3

$$p_G(d_{1:3} \mid x_{1:3}) = \frac{\exp\left(\alpha \sum_{t=1}^{3} (\mathbb{1}_{(d_{t-1}, d_t) \in \mathcal{T}} + \mathbb{1}_{(x_t, d_t) \in \mathcal{E}})\right)}{Z_G(x_{1:3})} \tag{374}$$
$$p_L(d_{1:3} \mid x_{1:3}) = p_L(d_1 \mid x_1) p_L(d_2 \mid d_1, x_{1:2}) p_L(d_3 \mid d_{1:2}, x_{1:3}) \tag{375}$$

where I've written p_L as a product over its local CPDs because it reveals the core observation that the proof is based on: *for any given subsequence $(d_{1:t-1}, x_{1:t})$, the local CPD is constrained to satisfy $\sum_{d_t} p_L(d_t \mid d_{1:t-1}, x_{1:t}) = 1$*. With this, the following comparison of p_G and p_L for large α completes the proof of $P_G \nsubseteq P_L$:

$$\lim_{\alpha \to \infty} p_G(ABC \mid abc) = \lim_{\alpha \to \infty} p_G(ADE \mid abe) = 1 \tag{376}$$
$$p_L(ABC \mid abc) + p_L(ADE \mid abe) \leq 1 \quad (\forall \alpha) \tag{377}$$

$\therefore P_L \subsetneq P_G$.

[111] This is for conditional models only.
[112] Note that $\subset$ and $\subsetneq$ mean the same thing. Matter of notational preference/being explicit/etc.

Sutton et al., "An Introduction to Conditional Random Fields," (2012).

Graphical Modeling (2.1). Some notation. Denote factors as $\psi_a(\boldsymbol{y}_a)$ where $1 \leq a \leq A$ and A is the total number factors. $\boldsymbol{y}_a$ is an assignment to the subset $Y_a \subseteq Y$ of variables associated with ψ_a. The value returned by ψ_a is a non-negative scalar that can be thought of as a measure of how compatible the values $\boldsymbol{y}_a$ are with each other. Given a collection of subsets $\{Y_a\}_{a=1}^A$ of Y, an **undirected graphical model** is the set of all distributions that can be written as

$$p(y) = \frac{1}{Z} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a) \tag{378}$$

$$Z = \sum_{\boldsymbol{y}} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a) \tag{379}$$

for any choice of **factors** $\mathcal{F} = \{\psi_a\}$ that have $\psi_a(\boldsymbol{y}_a) \geq 0$ for all $\boldsymbol{y}_a$. The sum for the **partition function**, Z, is over all possible assignments $\boldsymbol{y}$ of the variables Y. We'll use the term ***random field*** to refer to a particular distribution among those defined by an undirected model[113].

We can represent the factorization with a **factor graph**: a bipartite graph $G = (V, F, E)$ in which one set of nodes $V = \{1, 2, \ldots, |Y|\}$ indexes the RVs in the model, and the set of nodes $F = \{1, 2, \ldots, A\}$ indexes the factors. A connection between a variable node Y_s for $s \in V$ to some factor node ψ_a for $a \in F$ means that Y_s is one of the arguments of ψ_a. It is common to draw the factor nodes as squares, and the variable nodes as circles.

Generative versus Discriminative Models (2.2). Naive Bayes is generative, while logistic regression (a.k.a maximum entropy) is discriminative. Recall that Naive Bayes and logistic are defined as, respectively,

$$p(y, \boldsymbol{x}) = p(y) \prod_{k=1}^{K} p(x_k \mid y) \tag{380}$$

$$p(y \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \exp\left(\sum_{k=1}^{K} \theta_k f_k(y, \boldsymbol{x})\right) \tag{381}$$

where the f_k in the definition of logistic regression denote the feature functions. We could set them, for example, as $f_{y',j}(y, \boldsymbol{x}) = 1_{y'=y} x_j$.

[113] i.e. a particular set of factors.

An example generative model for sequence prediction is the **HMM**. Recall that an HMM defines

$$p(\boldsymbol{y}, \boldsymbol{x}) = \prod_{t=1}^{T} p(y_t \mid y_{t-1}) p(x_t \mid y_t) \tag{382}$$

where we are using the dummy notation of assuming an initial-initial state y_0 clamped to 0 and begins every state sequence, so we can write the initial state distribution as $p(y_1 \mid y_0)$.

We see that the generative models, like naive Bayes and the HMM, define a family of joint distributions that factorizes as $p(y, x) = p(y)p(x \mid y)$. Discriminative models, like logistic regression, define a family of conditional distributions $p(y \mid x)$. The main conceptual difference here is that a conditional distribution $p(y \mid x)$ doesn't include a model of $p(x)$. The principal advantage of discriminative modeling is that it's better suited to include rich, overlapping features. Discriminative models like CRFs make conditional independence assumptions both (1) among y and (2) about how the y can depend on x, but do *not* make conditional independence assumptions among x.

The difference between NB and LR is due *only* to the fact that NB is generative and LR is discriminative. Any LR classifier can be converted into a NB classifier with the same decision boundary, and vice versa. In other words, NB defines the same family as LR, if we interpret NB generatively as

$$p(y, \boldsymbol{x}) = \frac{\exp\left(\sum_k \theta_k f_k(y, \boldsymbol{x})\right)}{\sum_{\widetilde{y}, \widetilde{x}} \exp\left(\sum_k \theta_k f_k(\widetilde{y}, \widetilde{x})\right)} \tag{383}$$

and train it to maximize the conditional likelihood. Similarly, if the LR model is interpreted as above, and trained to maximize the joint likelihood, then we recover the same classifier as NB.

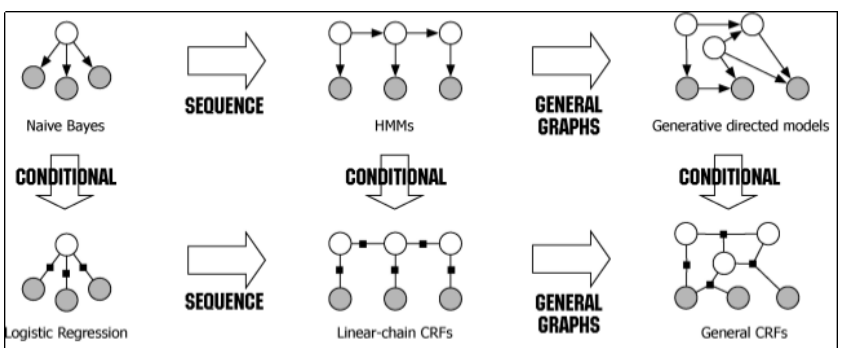


Fig. 2.4 Diagram of the relationship between naive Bayes, logistic regression, HMMs, linear-chain CRFs, generative models, and general CRFs.

Linear-Chain CRFs (2.3). Key point: the conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that follows from the joint distribution $p(\boldsymbol{y}, \boldsymbol{x})$ of an HMM is in fact a CRF with a particular choice of feature functions. First, we rewrite the HMM joint in a form that's more amenable to generalization:

$$p(\boldsymbol{y}, \boldsymbol{x}) = \frac{1}{Z} \prod_{t=1}^{T} \exp\left(\sum_{i,j \in S} \theta_{i,j} \mathbb{1}_{\{y_t = i, y_{t-1} = j\}} + \sum_{i \in S, o \in O} \mu_{o,i} \mathbb{1}_{\{y_t = i, x_t = o\}} \right) \tag{384}$$

HMM joint distribution

$$= \frac{1}{Z} \prod_{t=1}^{T} \exp\left(\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, x_t) \right) \tag{385}$$

and the latter provides the more compact notation[114]. We can use Bayes rule to then write $p(\boldsymbol{y} \mid \boldsymbol{x})$, which would give us a particular kind of linear-chain CRF that only includes features for the current word's identity. The general definition of linear-chain CRFs is given below:

Let Y, X be random vectors, $\theta = \{\theta_k\} \in \mathbb{R}^K$ be a parameter vector, and $\mathcal{F} = \{f_k(y, y', \boldsymbol{x}_t)\}_{k=1}^K$ be a set of real-valued feature functions. Then a ***linear-chain conditional random field*** *is a distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that takes the form:*

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t) \right) \tag{386}$$

$$Z(\boldsymbol{x}) = \sum_{\boldsymbol{y}} \prod_{t=1}^{T} \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t) \right) \tag{387}$$

Notice that a linear chain CRF can be described as a factor graph over $\boldsymbol{x}$ and $\boldsymbol{y}$, where each local function (factor) ψ_t has the special log-linear form:

$$\psi_t(y_t, y_{t-1}, x_t) = \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, x_t) \right) \tag{388}$$

General CRFs (2.4). Let G be a factor graph over X and Y. Then (X, Y) is a conditional random field if for any value $\boldsymbol{x}$ of X, the distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ factorizes according to G. If $F = \{\psi_a\}$ is the set of A factors in G, then the conditional distribution for a CRF is

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a, \boldsymbol{x}_a) \tag{389}$$

It is often useful to require that the factors be log-linear over a prespecified set of feature functions, which allows us to write the conditional distribution as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{\psi_a \in F} \exp\left(\sum_{k=1}^{K(a)} \theta_{a,k} f_{a,k}(\boldsymbol{y}_a, \boldsymbol{x}_a) \right) \tag{390}$$

[114] Note how we collapsed the summations over i, j and i, o to simply k. This is purely notational. Each value k can be mapped to/from a unique i, j or i, o in the first version. Also note that, necessarily, each feature function f_k in the latter version maps to a specific indicator function $\mathbb{1}$ in the first.

In addition, most models rely extensively on **parameter tying**[115]. To denote this, we partition the factors of G into $\mathcal{C} = \{C_1, C_2, \ldots, C_P\}$, where each C_p is a **clique template**: a set of factors sharing a set of feature functions $\{f_{p,k}(\boldsymbol{x}_c, \boldsymbol{y}_c)\}_{k=1}^{K(p)}$ and a corresponding set of parameters $\boldsymbol{\theta}_p \in \mathbb{R}^{K(p)}$. A CRF that uses clique templates can be written as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{C_p \in \mathcal{C}} \prod_{\psi_c \in C_p} \psi_c(\boldsymbol{x}_c, \boldsymbol{y}_c; \boldsymbol{\theta}_p) \tag{391}$$

$$= \frac{1}{Z(\boldsymbol{x})} \prod_{C_p \in \mathcal{C}} \prod_{c \in C_p} \exp\left\{ \sum_{k=1}^{K(p)} \theta_{p,k} f_{p,k}(\boldsymbol{x}_c, \boldsymbol{y}_c) \right\} \tag{392}$$

In a linear-chain CRF, typically one uses one clique template $C_0 = \{\psi_t\}_{t=1}^{T}$. Again, each factor in a given template *shares the same feature functions and parameters*, so the previous sentence means that we reuse the set of features and parameters for each timestep.

Feature engineering (2.5).

- **Label-observation features**. When our label variables are discrete, the features $f_{p,k}$ of a clique template C_p are ordinarily chosen to have a particular form:

$$f_{p,k}(\boldsymbol{y}_c, \boldsymbol{x}_c) = 1_{\{\boldsymbol{y}_c = \widetilde{\boldsymbol{y}}_c\}} q_{p,k}(\boldsymbol{x}_c) \tag{393}$$

  and we refer to the functions $q_{p,k}(\boldsymbol{x}_c)$ as *observation functions*.
- **Unsupported features**. Many observation-label pairs may never occur in our training data (e.g. having the word "with" being associated with label "CITY"). Such features are called unsupported features, and can be useful since often their weights will be driven negative, which can help prevent the model from making predictions in the future that are far from what was seen in the training data.
- **Edge-Observation and Node-Observation features**: the two most common types of label-observation features. Edge-observation features are for the transition factors, while node-observation features are the form introduced for label-observation features above.

$$\text{[edge-obs]} \quad f(y_t, y_{t-1}, \boldsymbol{x}_t) = q_m(\boldsymbol{x}_t) \mathbb{1}_{y_t = y, y_{t-1} = y'} \quad (\forall y, y' in \mathcal{Y}, \forall m) \tag{394}$$

$$\text{[node-obs]} \quad f(y_t, y_{t-1}, \boldsymbol{x}_t) = \mathbb{1}_{y_t = y, y_{t-1} = y'} \quad (\forall y, y' in \mathcal{Y}) \tag{395}$$

  and both use the same $f(y_t, \boldsymbol{x}_t) = q_m(\boldsymbol{x}_t) \mathbb{1}_{y_t = y}$ $(\forall y, \in \mathcal{Y}, \forall m)$. Recall that m is the index into our set of observation features[116].
- **Feature Induction**. The model begins with a number of base features, and the training procedure adds conjunctions of those features.

[115] Note how, for CRFs, the actual parameters θ are tightly coupled with the feature functions f.

[116] In CRFSuite, the observation features are all the attributes we define, and any features that use both label and observation are defined within CRFSuite itself.

Inference (Sec. 4)

There are two inference problems that arise:

1. Wanting to predict the labels of a new input $\boldsymbol{x}$ using the most likely labeling $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x})$.
2. Computing marginal distributions (during parameter estimation, for example) such as node marginals $p(y_t \mid \boldsymbol{x})$ and edge marginals $p(y_t, y_{t-1} \mid \boldsymbol{x})$.

For linear-chain CRFs, the **forward-backward algorithm** is used for computing marginals, and the **Viterbi algorithm** for computing the most probable assignment. We'll first derive these for the case of HMMs, and then generalize to the linear-chain CRF case.

Forward-backward algorithm (HMMs). An efficient technique for computing marginals. We begin by writing out $p(\boldsymbol{x})$, and using the distributive law to convert the sum of products to a product of sums:

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{y}} p(\boldsymbol{x}, \boldsymbol{y}) \tag{396}$$

We define the ψ_t as $p(y_t \mid y_{t-1})p(x_t \mid y_t)$

$$= \sum_{\boldsymbol{y}} \prod_{t=1}^{T} \psi_t(y_t, y_{t-1}, x_t) \tag{397}$$

$$= \sum_{y_1} p(y_1)p(x_1 \mid y_1) \sum_{y_2} p(y_2 \mid y_1)p(x_2 \mid y_2) \sum_{y_3} \cdots \sum_{y_T} p(y_T \mid y_{T-1})p(x_T \mid y_T) \tag{398}$$

$$= \sum_{y_1} \psi_1(y_1, x_1) \sum_{y_2} \cdots \sum_{y_T} \psi_T(y_T, y_{T-1}, x_T) \tag{399}$$

$$= \sum_{T} \sum_{T-1} \psi_T(y_T, y_{t-1}, x_T) \sum_{y_{T-2}} \cdots \sum_{y_1} \psi_1(y_1, x_1) \tag{400}$$

We see that we can save an exponential amount of work by caching the inner sums as we go. Let M denote the number of possible states for the y variables. We define a set of T **forward variables** α_t, each of which is a vector of size M:

$$\alpha_t(j) \triangleq p(\boldsymbol{x}_{\langle 1 \ldots t \rangle}, y_t = j) \tag{401}$$

$$= \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} p(\boldsymbol{x}_{\langle 1 \ldots t \rangle}, \boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}, y_t = j) \tag{402}$$

$$= \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \psi_t(j, y_{t-1}, x_t) \prod_{t'=1}^{t'-1} \psi_{t'}(y_{t'}, y_{t-1}, x_{t'}) \tag{403}$$

$$= \sum_{i \in S} \psi_t(j, i, x_t) \sum_{\boldsymbol{y}_{\langle 1 \ldots t-2 \rangle}} \psi_{t-1}(y_{t-1}, y_{t-2}, x_{t-1}) \prod_{t'=1}^{t-2} \psi_{t'}(y_{t'}, y_{t-1}, x_{t'}) \tag{404}$$

$$= \sum_{i \in S} \psi_t(j, i, x_t) \alpha_{t-1}(i) \tag{405}$$

where S is the set of M possible states. By recognizing that $p(\boldsymbol{x}) = \sum_{j \in S} \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} p(\boldsymbol{x}_{\langle 1 \ldots t \rangle}, \boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}, j)$, we can rewrite $p(\boldsymbol{x})$ as

$$p(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) \tag{406}$$

Notice how in the step from equation 401 to 402, we marginalized over all possible y subsequences that could've been aligned with $\boldsymbol{x}_{\langle 1 \ldots t \rangle}$. We will repeat this pattern to derive the backward recursion for β_t, which is the same idea except now we go from T backward until some t (instead of going from 1 *forward* until some t).

$$\begin{aligned}
\beta_t(i) &\triangleq p(\boldsymbol{x}_{\langle t+1 \ldots T \rangle} \mid y_t = i) && (407) \\
&= \sum_{\boldsymbol{y}_{\langle t+1 \ldots T \rangle}} \psi_{t+1}(y_{t+1}, i, x_{t+1}) \prod_{t'=t+2}^{T} \psi_{t'}(y_{t'}, y_{t'-1}, x_{t'}) && (408) \\
&= \sum_{y_{t+1}} \psi_{t+1}(y_{t+1}, i, x_{t+1}) \beta_{t+1}(y_{t+1}) && (409)
\end{aligned}$$

We initialize $\beta_T(i) = 1$.

Similar to how we obtained equation 406, we can rewrite $p(\boldsymbol{x})$ in terms of the β:

$$p(\boldsymbol{x}) = \beta_0(y_0) \triangleq \sum_{y_1} \psi_1(y_1, y_0, x_1) \beta_1(y_1) \tag{410}$$

We can then combine the definition for α and β to compute marginals of the form $p(y_{t-1}, y_t \mid \boldsymbol{x})$:

$$p(y_{t-1}, y_t \mid \boldsymbol{x}) = \frac{1}{p(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{411}$$

$$\text{where} \quad p(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{412}$$

In summary, the forward-backward algorithm consists of the following steps:

1. Compute α_t for all t using equation 406.
2. Compute β_t for all t using equation 410.
3. Return the marginal distributions computed from equation 411.

Viterbi algorithm (HMMs). For computing $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x})$. The derivation is nearly the same as how we derived the forward-backward algorithm, except now we've replaced the summations in equation 400 with maximization. The analog of α for viterbi are defined as:

$$\delta_t(j) = \max_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \psi_t(j, y_{t-1}, x_t) \prod_{t'=1}^{t-1} \psi_{t'}(y_{t'}, y_{t'-1}, x_{t'}) \tag{413}$$

$$= \max_{i \in S} \psi_t(j, i, x_t) \delta_{t-1}(i) \tag{414}$$

and the maximizing assignment is computed by a backwards recursion,

$$y_T^* = \arg\max_{i \in S} \delta_T(i) \tag{415}$$

$$y_t^* = \arg\max_{i \in S} \psi_t(y_{t+1}^*, i, x_{t+1}) \delta_t(i) \text{ for } t < T \tag{416}$$

Computing the recursions for δ_t and y_t^* together is the *Viterbi algorithm*.

Forward-backward and Viterbi for linear-chain CRF. Generalizing to the linear-chain CRF, where now

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \psi_t(y_t, y_{t-1}, x_t) \tag{417}$$

$$\text{where} \quad \psi_t(y_t, y_{t-1}, x_t) = \exp\left\{ \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{418}$$

and the results for the forward-backward algorithm become

$$p(y_{t-1}, y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{419}$$

$$p(y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_t(y_t) \beta_t(y_t) \tag{420}$$

$$\text{where} \quad Z(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{421}$$

Note that the interpretation is also slightly different. The α, β, and δ variables should only be interpreted with the factorization formulas, and *not* as probabilities. Specifically, use

$$\alpha_t(j) = \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \exp\left\{ \sum_k \theta_k f_k(j, y_{t-1}, x_t) \right\} \prod_{t'=1}^{t'-1} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{422}$$

$$\beta_t(i) = \sum_{\boldsymbol{y}_{\langle t+1 \ldots T \rangle}} \exp\left\{ \sum_k \theta_k f_k(y_{t+1}, i, x_{t+1}) \right\} \prod_{t'=t+2}^{T} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{423}$$

$$\delta_t(j) = \max_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \exp\left\{ \sum_k \theta_k f_k(j, y_{t-1}, x_t) \right\} \prod_{t'=1}^{t-1} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{424}$$

Markov Chain Monte Carlo (MCMC). The two most popular classes of approximate inference algorithms are **Monte Carlo** algorithms and **variational** algorithms. In what follows, we drop the CRF-specific notation and refer to the more general joint distribution

$$p(\boldsymbol{y}) = Z^{-1} \prod_{a \in F} \psi_a(\boldsymbol{y}_a) \tag{425}$$

that factorizes according to some factor graph $G = (V, F)$. MCMC methods construct a Markov chain whose state space is the same as that of Y, and sample from this chain to approximate, e.g., the expectation of some function $f(\boldsymbol{y})$ over the distribution $p(\boldsymbol{y})$. MCMC algorithms aren't commonly applied in the context of CRFs, since parameter estimation by maximum likelihood requires calculating marginals many times.

Parameter Estimation (Sec. 5)

Maximum Likelihood for Linear-Chain CRFs. Since we're modeling the conditional distribution with CRFs, we use the **conditional log likelihood** $\ell(\boldsymbol{\theta})$ with l2-regularization:

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - \frac{1}{2\sigma^2} \sum_{k=1}^{K} \theta_k^2 \tag{426}$$

$$= \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{K} \theta_k f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i=1}^{N} \log Z(\boldsymbol{x}^{(i)}) - \frac{1}{2\sigma^2} \sum_{k=1}^{K} \theta_k^2 \tag{427}$$

with regularization parameter $1/2\sigma^2$. The partial derivatives are

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{i,t} f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i,t,y,y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{\sigma^2} \tag{428}$$

and the derivation for the partial derivative of $\log(Z(x))$ is

$$\frac{\partial \log Z(x)}{\partial \theta_k} = \frac{1}{Z(x)} \frac{\partial}{\partial \theta_k} \left[\sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \prod_{t=1}^{T} \exp \left\{ \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \right] \tag{429}$$

$$= \frac{1}{Z(x)} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \frac{\partial}{\partial \theta_k} \exp \left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{430}$$

$$= \frac{1}{Z(x)} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \sum_t f_k(y_t, y_{t-1}, x_t) \exp \left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{431}$$

$$= \sum_t \sum_{y_t} \sum_{y_{t-1}} f_k(y_t, y_{t-1}, x_t) \left[\sum_{\boldsymbol{y}_{\langle 1 \ldots t-2 \rangle}} \sum_{\boldsymbol{y}_{\langle t+1 \ldots T \rangle}} \frac{1}{Z(x)} \exp \left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \right] \tag{432}$$

$$= \sum_t \sum_y \sum_{y'} f_k(y, y', x_t) p(y_t = y, y_{t-1} = y' \mid x) \tag{433}$$

We can rewrite this in the form of expectations. For now, let $\widetilde{p}(\boldsymbol{y}, \boldsymbol{x})$ denote the *empirical distribution*, and let $\hat{p}(\boldsymbol{y} \mid \boldsymbol{x}; \theta)\widetilde{p}(\boldsymbol{x})$ denote the *model distribution*.

$$\frac{\partial \ell}{\partial \theta_k} = \mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim \widetilde{p}(\boldsymbol{y}, \boldsymbol{x})} \left[\sum_t f_k(y_t, y_{t-1}, x_t) \right] - \mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim \hat{p}(\boldsymbol{y}, \boldsymbol{x})} \left[\sum_t f_k(y_t, y_{t-1}, x_t) \right] \tag{434}$$

Procedure: Training Linear-Chain CRFs

Here I summarize the main steps involved during a parameter update.

Inference. We need to compute the log probabilities for each instance in the dataset, under the current parameters. We will need them when evaluating $Z(\boldsymbol{x})$ and the marginals $p(y, y' \mid \boldsymbol{x})$ when computing gradients.

1. Initialize $\alpha_1(j) = \exp\{\sum_k \theta_k f_k(j, y_0, x_1)\}$ (y_0 is the fixed initial state) and $\beta_T(i) = 1$.
2. For all t, compute:

$$\alpha_t(j) = \sum_{i \in S} \exp\left\{ \sum_k \theta_k f_k(j, i, x_t) \right\} \alpha_{t-1}(i) \tag{435}$$

$$\beta_t(j) = \sum_{i \in S} \exp\left\{ \sum_k \theta_k f_k(i, j, x_{t+1}) \right\} \beta_{t+1}(i) \tag{436}$$

3. Compute the marginals:

$$p(y_t, y_{t-1}, \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{437}$$

$$p(y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_t(y_t) \beta_t(y_t) \tag{438}$$

$$\text{where} \quad Z(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{439}$$

Gradients. For each parameter θ_k, compute:

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{i,t} f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i,t,y,y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{\sigma^2} \tag{440}$$

CRF with latent variables. Now we have additional latent variables $\boldsymbol{z}$:

$$p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{441}$$

Since we observe $\boldsymbol{y}$ during training, what if we instead treat this as a CRF with both $\boldsymbol{x}$ *and* $\boldsymbol{y}$ observed?

$$p(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{y}, \boldsymbol{x})} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{442}$$

$$Z(\boldsymbol{y}, \boldsymbol{x}) = \sum_{\boldsymbol{z}} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{443}$$

We can use the same inference algorithms as usual to compute $Z(\boldsymbol{y}, \boldsymbol{x})$, and the key result is that we can now write

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \sum_{\boldsymbol{z}} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) = \frac{Z(\boldsymbol{y}, \boldsymbol{x})}{Z(\boldsymbol{x})} \tag{444}$$

Finally, we can write the gradient as[117]

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{\boldsymbol{z}} p(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{x}) \frac{\partial}{\partial \theta_k} [\log p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x})] \tag{445}$$

$$= \sum_t \sum_{\boldsymbol{z}_t} \left[p(\boldsymbol{z}_t \mid \boldsymbol{y}, \boldsymbol{x}) f_k(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) - \sum_{y,y'} p(\boldsymbol{z}_t, y, y' \mid \boldsymbol{x}_t) f_k(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \right] \tag{446}$$

where I've assumed there are no connections between any $\boldsymbol{z}_t$ and $\boldsymbol{z}_{t' \neq t}$.

Stochastic Gradient Methods. Now we compute gradients for a single example, and for a linear-chain CRF:

$$\frac{\partial \ell_i}{\partial \theta_k} = \sum_t f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{t,y,y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{N\sigma^2} \tag{447}$$

which corresponds to parameter update (remember: we are using the LL, not the negative LL):

$$\theta^{(m)} = \theta^{(m-1)} + \alpha_m \nabla \ell_i(\theta^{(m-1)}) \tag{448}$$

where m denotes this is the mth update of the training process.

[117] This uses the trick

$$\frac{df}{d\theta} = f(\theta) \frac{d \log f}{d\theta}$$

MEMMs. Maximum-entropy Markov models. Essentially a Markov model in which the transition probabilities are given by logistic regression. Formally, a MEMM is defined by

$$p_{MEMM}(\boldsymbol{y} \mid \boldsymbol{x}) = \prod_{t=1}^{T} p(y_t \mid y_{t-1}, \boldsymbol{x}) \tag{449}$$

$$p(y_t \mid y_{t-1}, \boldsymbol{x}) = \frac{\exp\left\{\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t)\right\}}{Z_t(y_{t-1}, \boldsymbol{x})} \tag{450}$$

$$Z_t(y_{t-1}, \boldsymbol{x}) = \sum_{y'} \exp\left\{\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t)\right\} \tag{451}$$

which has some important differences compared to the linear-chain CRF. Notice how maximum-likelihood training of MEMMs does *not* require performance inference over full output sequences $\boldsymbol{y}$, because Z_t is a simple sum over the labels at a single position. MEMMs, however, suffer from *label bias*, while CRFs do not.

Bayesian CRFs. Instead of predicting the optimal labeling y^*_{ML} for input sequence x with maximum likelihood (ML), we can instead use a fully Bayesian (B) approach, both of which are shown below for comparison:

$$y^*_{ML} \leftarrow arg\, max_y p(y \mid x; \hat{\theta}) \tag{452}$$

$$y^*_{B} \leftarrow arg\, max_y \mathbb{E}_{\theta \sim p(\theta|x)}(p(y \mid x; \theta)) \tag{453}$$

Unfortunately, computing the exact integral (the expectation) is usually intractable, and we must resort to approximate methods like MCMC.

Co-sampling: Training Robust Networks for Extremely Noisy Supervision

Written by Brandon McKinzie

Han et al., "Co-sampling: Training Robust Networks for Extremely Noisy Supervision," (2018).

Introduction. The authors state that current methodologies [for training networks under noisy labels] involves estimating the **noise transition matrix** (which they don't define). Patrini et al. (2017) define the matrix as follows:

> *Denote by $T \in [0,1]^{c \times c}$ the noise transition matrix specifying the probability of one label being flipped to another, so that $\forall i, j \; T_{ij} \triangleq Pr\left[\widetilde{y} = e^j \mid y = e^i\right]$. The matrix is row-stochastic*[118] *and not necessarily symmetric across the classes.*

Algorithm. Authors propose a learning paradigm called **Co-sampling**. They maintain two networks f_{w_1} and f_{w_2} simultaneously. For each mini-batch data $\hat{\mathcal{D}}$, each network selects $\mathcal{R}_T$ small-loss instances as a "clean" mini-batch $\hat{\mathcal{D}}_1$ and $\hat{\mathcal{D}}_2$, respectively. Each of the two networks then uses the clean mini-batch data to update the parameters w_2 (w_1) of its *peer* network.

- Why small-loss instances? Because deep networks tend to fit clean instances first, then noisy/harder instances progressively after.
- Why two networks? Because if we just trained a single network on clean instances, we would not be robust in extremely high-noise rates, since the training error would accumulate if the selected instances are not "fully clean."

The Co-sampling paradigm algorithm is shown below.

Input: w_1 and w_2; learning rate η; fixed $\mathfrak{D}$; epoch T_k and T_{max}; iteration N_{max};
for $T = 1, 2, \ldots, T_{max}$ **do**
  Shuffle: training set $\widetilde{\mathcal{D}}$ //noisy dataset
  for $N = 1, \ldots, N_{max}$ **do**
    Draw: mini-batch $\widehat{\mathcal{D}}$ from $\widetilde{\mathcal{D}}$
    Sample: $\widehat{\mathcal{D}}_1 = \arg\min_{\widehat{\mathcal{D}}} \ell(f_{w_1}, \widehat{\mathcal{D}}, \mathfrak{R}_T)$ //sample $\mathfrak{R}_T$ small-loss instances
    Sample: $\widehat{\mathcal{D}}_2 = \arg\min_{\widehat{\mathcal{D}}} \ell(f_{w_2}, \widehat{\mathcal{D}}, \mathfrak{R}_T)$ //sample $\mathfrak{R}_T$ small-loss instances
    Update: $w_1 = w_1 - \eta \nabla f_{w_1}(\widehat{\mathcal{D}}_2)$ //update w_1 by $\widehat{\mathcal{D}}_2$
    Update: $w_2 = w_2 - \eta \nabla f_{w_2}(\widehat{\mathcal{D}}_1)$ //update w_2 by $\widehat{\mathcal{D}}_1$
  end
  Update: $\mathfrak{R}_T = 1 - \min\{\frac{T}{T_k}\mathfrak{D}, \mathfrak{D}\}$
end
Output: f_{w_1} **and** f_{w_2}

[118] Each row sums to 1.

Hidden-Unit Conditional Random Fields

 Written by Brandon McKinzie

Maaten et al., "Hidden-Unit Conditional Random Fields," (2011).

Introduction. Three key advantages of CRFs over HMMs:

1. CRFs don't assume that the observations are conditionally independent given the hidden (or target if linear-chain CRF) states.
2. CRFs don't suffer from the label bias problems of models that do local probability normalization[119].
3. For certain choices of factors, the negative conditional L.L. is convex.

The hidden-unit CRF (HUCRF), similar to discriminative restricted Boltzmann machines (RBMs), has binary stochastic hidden units that are conditionally independent given the data and the label sequence. By exploiting the conditional independence properties, we can efficiently compute:

1. The exact gradient of the C.L.L.
2. The most likely label sequence.
3. The marginal distributions over label sequences.

Hidden-Unit CRFs. At each time step t, the HUCRF employs H binary stochastic hidden units $\boldsymbol{z}_t$. It models the conditional distribution as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{\boldsymbol{Z}(\boldsymbol{x})} \sum_{\boldsymbol{z}} \exp\left(E(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y})\right) \tag{454}$$

$$\begin{aligned} E(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y}) = &\sum_{t=2}^{T} \left[\boldsymbol{y}_{t-1}^T \boldsymbol{A} \boldsymbol{y}_t\right] + \sum_{t=1}^{T} \left[\boldsymbol{x}_t^T \boldsymbol{W} \boldsymbol{z}_t + \boldsymbol{z}_t^T \boldsymbol{V} \boldsymbol{y}_t + \boldsymbol{b}^T \boldsymbol{z}_t + \boldsymbol{c}^T \boldsymbol{y}_t\right] \\ &+ \boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau} \end{aligned} \tag{455}$$

Since the hidden units are conditionally independent given the data and labels, the hidden units can be marginalized out one-by-one. This, along with the nice property that the hidden units have binary elements, allows us to write $p(\boldsymbol{y} \mid \boldsymbol{x})$ without writing any $\boldsymbol{z}_t$ explicitly, as

[119] See the introduction in my CRF notes for recap of label-bias.

shown below:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \Bigg[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \sum_{z_h \in \{0,1\}} \exp\{z_h b_h + z_h \boldsymbol{w}_h^T \boldsymbol{x}_t + z_h \boldsymbol{v}_h^T \boldsymbol{y}_t\} \Bigg] \tag{456}$$

$$= \frac{\exp\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \Bigg[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \Big(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\Big) \Bigg] \tag{457}$$

For inference, we'll need an algorithm for computing the marginals $p(y_t \mid \boldsymbol{x})$ and $p(y_t, y_{t-1} \mid \boldsymbol{x})$. The equations are essentially the same as the forward-backward formulas for the linear-chain CRF, but with summations over $\boldsymbol{z}$:

$$p(y_t, y_{t-1} \mid \boldsymbol{x}) \propto \alpha_{t-1}(y_{t-1}) \Bigg[\sum_{\boldsymbol{z}_t} \Psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, y_t, y_{t-1}) \Bigg] \beta_t(y_t) \tag{458}$$

$$p(y_t \mid \boldsymbol{x}) \propto \alpha_t(y_t) \beta_t(y_t) \tag{459}$$

$$\tag{460}$$

$$\alpha_t(j) = \sum_{i \in \mathcal{Y}} \sum_{\boldsymbol{z}_t} \Psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, j, i) \alpha_{t-1}(i) \tag{461}$$

$$\beta_t(j) = \sum_{i \in \mathcal{Y}} \sum_{\boldsymbol{z}_{t+1}} \Psi_{t+1}(\boldsymbol{x}_{t+1}, \boldsymbol{z}_{t+1}, i, j) \beta_{t+1}(i) \tag{462}$$

Training. The conditional log likelihood for a single example $(\boldsymbol{x}, \boldsymbol{y})$ is (bias and initial-state terms omitted)

$$\mathcal{L} = \log p(\boldsymbol{y} \mid \boldsymbol{x}) \tag{463}$$

$$= \sum_{t=1}^{T} \log \left(\sum_{\boldsymbol{z}_t} \Psi_t \left(\boldsymbol{x}_t, \boldsymbol{z}_t, y_{t-1}, y_t \right) \right) - \log Z(\boldsymbol{x}) \tag{464}$$

$$\text{where} \qquad \Psi_t := \exp\{y_{t-1} \boldsymbol{A} y_t + \boldsymbol{x}_t^T \boldsymbol{W} \boldsymbol{z}_t + \boldsymbol{z}_t^T \boldsymbol{V} y_t\} \tag{465}$$

Let $\Upsilon = \{\boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}, \boldsymbol{c}, \}$ be the set of model parameters. The gradient w.r.t. the data-dependent[120] parameters $v \in \Upsilon$ is given by

$$\frac{\partial \mathcal{L}}{\partial v} = \sum_{t=1}^{T} \left[\sum_{k \in \mathcal{Y}} \left((\mathbb{1}_{y_t = k} - p(y_t = k \mid \boldsymbol{x})) \sum_{h=1}^{H} \sigma(o_{hk}(\boldsymbol{x}_t)) \frac{\partial o_{hk}(\boldsymbol{x}_t)}{\partial v} \right) \right] \tag{466}$$

$$\text{where} \qquad o_{hk}(\boldsymbol{x}_t) = b_h + c_k + V_{hk} + \boldsymbol{w}_h^T \boldsymbol{x}_t \tag{467}$$

Unfortunately, the negative CLL is **non-convex**, and so we are only guaranteed to converge to a *local* maximum of the CLL.

[120] The data-dependent parameters are each individual element of the elements of Υ. "Data" here means $(\boldsymbol{x}, \boldsymbol{y})$. Notice that Υ does not include $\boldsymbol{A}$, $\boldsymbol{\pi}$, or $\boldsymbol{\tau}$.

Detailed Derivations

Unfortunately, the paper leaves out a lot of details regarding derivations and implementations. I'm going to work through them here. First, a recap of the main equations, and with all biases/initial states included. Not leaving anything out[121]

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\left\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\right\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \left[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] \tag{468}$$

$$NLL = -\sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) \tag{469}$$

$$= -\sum_{i=1}^{N} \left[\sum_{t=1}^{T} \log \left(\sum_{\boldsymbol{z}_t} \psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, \boldsymbol{y}_{t-1}, \boldsymbol{y}_t) \right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{470}$$

The above formula for $p(\boldsymbol{y} \mid \boldsymbol{x})$ implies something that will be very useful:

$$\sum_{\boldsymbol{z}_t} \psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, \boldsymbol{y}_{t-1}, \boldsymbol{y}_t) = \exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \tag{471}$$

Using the generalization of the product rule for derivatives over N products, we can derive that

$$\frac{\partial}{\partial v} \prod_h (1 + \exp\{o(h)\}) = \left[\prod_h (1 + \exp\{o(h)\}) \right] \left[\sum_h \sigma\left(o\left(h\right)\right) \frac{\partial o(h)}{\partial v} \right] \tag{472}$$

Which means the derivatives of $\sum_z \psi$ for the data-dependent params v_{dat} and transition params v_{tr}, are:

$$\frac{\partial \sum_{\boldsymbol{z}_t} \psi_t}{\partial v_{dat}} = \left[\sum_h \sigma\left(o\left(h, y_t\right)\right) \frac{\partial o(h, y_t)}{\partial v_{dat}} \right] \sum_{\boldsymbol{z}_t} \psi_t \tag{473}$$

$$\frac{\partial \sum_{\boldsymbol{z}_t} \psi_t}{\partial v_{tr}} = \left[\frac{\partial}{\partial v_{tr}} \boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t \right] \sum_{\boldsymbol{z}_t} \psi_t \tag{474}$$

which also conveniently means that

$$\frac{\partial}{\partial v_{dat}} \log \left(\sum_{\boldsymbol{z}_t} \psi_t \right) = \sum_h \sigma\left(o\left(h, y_t\right)\right) \frac{\partial o(h, y_t)}{\partial v_{dat}} \tag{475}$$

$$\frac{\partial}{\partial v_{tr}} \log \left(\sum_{\boldsymbol{z}_t} \psi_t \right) = \frac{\partial}{\partial v_{tr}} \boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t \tag{476}$$

I'll now proceed to derive the gradients of negative (conditional) log-likelihood for the main parameters. We can save some time by getting the base formula for any of the gradients with

[121] The equation for $p(\boldsymbol{y} \mid \boldsymbol{x})$ from the paper, and thus here, is technically incorrect. The term $\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\}$ should not be included in the product over t for $t = 1$.

respect to a specific single parameter v:

$$\frac{\partial NLL}{\partial v} = -\sum_{i=1}^{N}\left[\sum_{t=1}^{T}\frac{\partial}{\partial v}\log\left(\sum_{\boldsymbol{z}_t}\psi_t(\boldsymbol{x}_t^{(i)},\boldsymbol{z}_t,\boldsymbol{y}_{t-1}^{(i)},\boldsymbol{y}_t^{(i)})\right) - \frac{\partial}{\partial v}\log Z(\boldsymbol{x}^{(i)})\right] \tag{477}$$

$$\frac{\partial \log Z(\boldsymbol{x}^{(i)})}{\partial v} = \frac{1}{Z(\boldsymbol{x}^{(i)})}\frac{\partial}{\partial v}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\widetilde{p}(\boldsymbol{y}_1,\ldots,\boldsymbol{y}_T \mid \boldsymbol{x}^{(i)}) \tag{478}$$

$$\frac{\partial}{\partial v}\widetilde{p}(\boldsymbol{y}_1,\ldots,\boldsymbol{y}_T \mid \boldsymbol{x}^{(i)}) = \frac{\partial}{\partial v}\prod_t\sum_{\boldsymbol{z}_t}\psi_t \tag{479}$$

$$= \left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\frac{\partial}{\partial v}\sum_{\boldsymbol{z}_t}\psi_t}{\sum_{\boldsymbol{z}_t}\psi_t}\right] \tag{480}$$

where I've done some regrouping on the last line to be more gradient-friendly.

Data-dependent parameters

All params v that are not transition params.

$$\frac{\partial NLL}{\partial v_{dat}} = -\sum_{i=1}^{N}\left[\sum_{t=1}^{T}\frac{\partial}{\partial v_{dat}}\log\left(\sum_{\boldsymbol{z}_t}\psi_t\right) - \frac{\partial}{\partial v}\log Z(\boldsymbol{x}^{(i)})\right] \tag{481}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\frac{\partial}{\partial v}\sum_{\boldsymbol{z}_t}\psi_t}{\sum_{\boldsymbol{z}_t}\psi_t}\right]\right] \tag{482}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\right] \tag{483}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_t\sum_y\sum_{y'}\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\xi_{t,y,y'}\right] \tag{484}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_t\sum_y\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\gamma_{t,y}\right] \tag{485}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\left(\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_y\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\gamma_{t,y}\right)\right] \tag{486}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_y\left(\left(\mathbb{1}_{y_t=y} - \gamma_{t,y}\right)\sum_h\sigma\left(o(h,y_t)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right)\right] \tag{487}$$

NOTE: Although I haven't thoroughly checked the last few steps, they are required to be true in order to match the paper's results.

Transition parameters

$$\frac{\partial NLL}{\partial v_{tr}} = -\sum_{i=1}^{N}\left[\sum_{t=1}^{T}\frac{\partial}{\partial v_{tr}}\log\left(\sum_{\boldsymbol{z}_t}\psi_t\right) - \frac{\partial}{\partial v_{tr}}\log Z(\boldsymbol{x}^{(i)})\right] \tag{488}$$

$$= -\sum_{i}^{N}\left[\sum_{t}\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right] - \frac{\partial}{\partial v_{tr}}\log Z(\boldsymbol{x}^{(i)})\right] \tag{489}$$

$$= -\sum_{i=1}^{N}\left[\sum_{t}\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right] - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right]\right] \tag{490}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_y\left((\mathbb{1}_{y_t=y} - \gamma_{t,y})\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right)\right] \tag{491}$$

Boundary parameters

$$\frac{\partial NLL}{\partial \pi_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_1=\ell} - \gamma_{1,\ell}\right] \tag{492}$$

$$\frac{\partial NLL}{\partial \tau_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_T=\ell} - \gamma_{T,\ell}\right] \tag{493}$$

The results of each of the boxes above are summarized below, for the case of $N = 1$ to save space.

$$\frac{\partial NLL}{\partial v_{dat}} = -\sum_t\sum_y\left((\mathbb{1}_{y_t=y} - \gamma_{t,y})\sum_h\sigma\left(o(h,y)\right)\frac{\partial o(h,y)}{\partial v_{dat}}\right) \tag{494}$$

$$\frac{\partial NLL}{\partial v_{tr}} = -\sum_t\sum_y\left((\mathbb{1}_{y_t=y} - \gamma_{t,y})\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right) \tag{495}$$

$$\frac{\partial NLL}{\partial \pi_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_1=\ell} - \gamma_{1,\ell}\right] \tag{496}$$

$$\frac{\partial NLL}{\partial \tau_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_T=\ell} - \gamma_{T,\ell}\right] \tag{497}$$

Now I'll further go through and show how the equations simplify for each type of data-

dependent parameter.

$$\frac{\partial NLL}{\partial W_{c,h}} = -\sum_t \sum_y \left((\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_{h'} \sigma\left(o(h',y)\right) \frac{\partial}{\partial W_{c,h}} \left(b_{h'} + c_y + V_{h',y} + \boldsymbol{w}_{h'}^T \boldsymbol{x}_t \right) \right) \tag{498}$$

$$= -\sum_t \sum_y \left((\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_{h'} \sigma\left(o(h,y)\right) \mathbb{1}_{h=h'} \mathbb{1}_{c \in \boldsymbol{x}_t} \right) \tag{499}$$

$$= -\sum_t \sum_y (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \, \sigma\left(o(h,y)\right) \mathbb{1}_{c \in \boldsymbol{x}_t} \tag{500}$$

$$\frac{\partial NLL}{\partial V_{h,y}} = -\sum_t (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \, \sigma\left(o(h,y)\right) \tag{501}$$

$$\frac{\partial NLL}{\partial b_h} = -\sum_t \sum_y (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \, \sigma\left(o(h,y)\right) \tag{502}$$

$$\frac{\partial NLL}{\partial c_y} = -\sum_t (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_h \sigma\left(o(h,y)\right) \tag{503}$$

$$\tag{504}$$

Alternative Approach. The above was a bit more cumbersome than needed. I'll now derive it in an easier way.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\left\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\right\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \left[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] \tag{505}$$

$$= \frac{\exp\{I+T\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \tag{506}$$

$$NLL = -\sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) \tag{507}$$

$$= -\sum_{i=1}^{N} \left[I + T + \sum_{t=1}^{T} \sum_{h=1}^{H} \left[\log\left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{508}$$

Now, focusing on the regular log-likelihood for a single example, we have

$$\frac{\partial \mathcal{L}_i}{\partial v} = \frac{\partial}{\partial v} \log p(\boldsymbol{y}^{(i)}, \boldsymbol{x}^{(i)}) \tag{509}$$

$$= \frac{\partial}{\partial v} \left[I + T + \sum_{t,h} \log\left(1 + \exp\left\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\right\}\right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{510}$$

$$\frac{\partial \log Z(\boldsymbol{x}^{(i)})}{\partial v} = \frac{1}{Z(\boldsymbol{x}^{(i)})} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \frac{\partial}{\partial v} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{511}$$

$$= \frac{1}{Z(\boldsymbol{x}^{(i)})} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \frac{\partial}{\partial v} \log \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{512}$$

as our base formula for partial derivatives.

Transition parameters

$$\frac{\partial \mathcal{L}_i}{\partial A_{i,j}} = \frac{\partial}{\partial A_{i,j}} \left[I + T + \sum_{t,h} \log \left(1 + \exp \left\{ b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t \right\} \right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{513}$$

$$= \frac{\partial}{\partial A_{i,j}} [I + T] - \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} p(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \frac{\partial}{\partial A_{i,j}} \left[y_1^T \pi + y_T^T \tau + \sum_t y_{t-1} A y_t \right] \tag{514}$$

$$= \sum_t \mathbb{1}_{y_{t-1}^{(i)}=i} \mathbb{1}_{y_t^{(i)}=j} - \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \frac{1}{Z(\boldsymbol{x}^{(i)})} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \sum_{t=1}^{T} \mathbb{1}_{y_{t-1}=i} \mathbb{1}_{y_t=j} \tag{515}$$

$$= \sum_t \mathbb{1}_{y_{t-1}^{(i)}=i} \mathbb{1}_{y_t^{(i)}=j} - \sum_t \sum_{y_t} \sum_{y_{t-1}} \mathbb{1}_{y_{t-1}=i} \mathbb{1}_{y_t=j} \sum_{\boldsymbol{y}_{\langle 1 \ldots t-2 \rangle}} \sum_{\boldsymbol{y}_{\langle t+1 \ldots T \rangle}} \frac{1}{Z(\boldsymbol{x}^{(i)})} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{516}$$

Pre-training of Hidden-Unit CRFs

Written by Brandon McKinzie

Kim et al., "Pre-training of Hidden-Unit CRFs," (2018).

Model Definition. The Hidden-Unit CRF (HUCRF) accepts the usual observation sequence $\boldsymbol{x} = x_1, \ldots, x_n$, and associated label sequence $\boldsymbol{y} = y_1, \ldots y_n$ for training. The HUCRF also has a hidden layer of binary-valued $\boldsymbol{z} = z_1 \ldots z_n$. It defines the joint probability

$$p_{\theta,\gamma}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{\exp\left(\boldsymbol{\theta}^T \Phi(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{\gamma}^T \Psi(\boldsymbol{z}, \boldsymbol{y})\right)}{\sum_{\boldsymbol{z}', \boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{x}, \boldsymbol{z}')} \exp\left(\boldsymbol{\theta}^T \Phi(\boldsymbol{x}, \boldsymbol{z}') + \boldsymbol{\gamma}^T \Psi(\boldsymbol{z}', \boldsymbol{y}')\right)} \tag{517}$$

where

- $\mathcal{Y}(\boldsymbol{x}, \boldsymbol{z})$ is the set of all possible label sequences for $\boldsymbol{x}$ and $\boldsymbol{z}$.
- $\Phi(\boldsymbol{x}, \boldsymbol{z}) = \sum_{j=1}^{n} \phi(x, j, z_j)$
- $\Psi(\boldsymbol{z}, \boldsymbol{y}) = \sum_{j=1}^{n} \psi(z_j, y_{j-1}, y_j)$.

Also note that we model $(z_i \perp z_{j \neq i} \mid \boldsymbol{x}, \boldsymbol{y})$.

Pre-training HUCRFs. Since the objective for HUCRFs is non-convex, we should choose a better initialization method than random initialization. This is where pre-training comes in, a simple 2-step approach:

1. Cluster observed tokens from M unlabeled sequences and treat the clusters as labels to train an intermediate HUCRF. Let $C(u^{(i)})$ be the sequence of cluster assignments/labels for the unlabeled sequence $u^{(i)}$. We compute:

$$(\theta_1, \gamma_1) \approx \arg\max_{\theta,\gamma} \sum_{i=1}^{M} \log p_{\theta,\gamma}(C(u^{(i)}) \mid u^{(i)}) \tag{518}$$

2. Train a final model on the labeled data $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, using θ_1 as an initialization point:

$$(\theta_2, \gamma_2) \approx \arg\max_{\theta,\gamma} \sum_{i=1}^{N} \log p_{\theta,\gamma}(y^{(i)} \mid x^{(i)}) \tag{519}$$

Note that pre-training only defines the initialization for θ, the parameters between $\boldsymbol{x}$ and $\boldsymbol{z}$. We still train γ, the parameters from $\boldsymbol{z}$ to $\boldsymbol{y}$, from scratch.

Canonical Correlation Analysis (CCA). A general technique that we will need to understand as a prerequisite for the multi-sense clustering approach (defined in the next section). Given n samples of the form $(x^{(i)}, y^{(i)})$, where each $x^{(i)} \in \{0,1\}^d$ and $y^{(i)} \in \{0,1\}^{d'}$, CCA returns *projection matrices* $A \in \mathbb{R}^{d \times k}$ and $B \in \mathbb{R}^{d' \times k}$ that we can use to project the samples to k dimensions:

$$x \longrightarrow A^T x \tag{520}$$

$$y \longrightarrow B^T y \tag{521}$$

The CCA algorithm is outlined below.

Algorithm: CCA

1. Calculate $\boldsymbol{D} \in \mathbb{R}^{d \times d'}$, $\boldsymbol{u} \in \mathbb{R}^d$, and $\boldsymbol{v} \in \mathbb{R}^{d'}$ as follows:

$$D_{i,j} = \sum_{l=1}^{n} \mathbb{1}_{x_i^{(l)}=1} \mathbb{1}_{y_j^{(l)}=1} \tag{522}$$

$$u_i = \sum_{l=1}^{n} \mathbb{1}_{x_i^{(l)}=1} \tag{523}$$

$$v_i = \sum_{l=1}^{n} \mathbb{1}_{y_i^{(l)}=1} \tag{524}$$

2. Define $\hat{\Omega} = \text{diag}(\boldsymbol{u})^{-1/2}\ \boldsymbol{D}\ \text{diag}(\boldsymbol{v})^{-1/2}$.
3. Calculate rank-k SVD $\hat{\Omega}$. Let $\boldsymbol{U} \in \mathbb{R}^{d \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{d' \times k}$ contain the left and right, respectively, singular vectors for the largest k singular values.
4. Return $A = \text{diag}(\boldsymbol{u})^{-1/2}\boldsymbol{U}$ and $B = \text{diag}(\boldsymbol{v})^{-1/2}\boldsymbol{V}$.

Multi-sense clustering. For each word type, use CCA to create a set of context embeddings corresponding to all occurrences of that word type. Then, cluster these embeddings with k-means. Set the number of word senses k to the number of label types occurring in the labeled data.

TODO: finish this note

Structured Attention Networks

Kim et al., "Structured Attention Networks," (2017).

Background: Attention Networks. Goal: produce a context c based on input sequence x and query q. We assume we have an attention distribution[122] $z \sim p(z \mid x, q)$. Interpret z as a categorical latent variable over T categories, where $T = len(x)$ is the length of the input sequence. We can then compute the context, $c = \mathbb{E}_{z \sim p(z|x,q)}[f(x,z)]$, where $f(x,z)$ is an *annotation function*[123].

> *We interpret the attention mechanism as taking the expectation of an annotation function $f(x,z)$ with respect to a latent variable $z \sim p$, where p is parameterized to be a function of x and q.*

For comparisons later on with the traditional attention mechanism, here it is:

$$c = \sum_{t}^{T} p(z = t \mid x, q)\boldsymbol{x}_t \tag{525}$$

$$p(z = t \mid x, q) = \text{softmax}(\theta_t) \tag{526}$$

where usually x is the sequence of hidden state of the encoder RNN, q is the hidden state of the decoder RNN at the most recent time step, z gives the source position to be attended to, and $\theta_t = \text{score}(x_t, q)$.

Structured Attention. In a **structured attention model**, z is now a *vector* of discrete random variables $z_1, \ldots, z_m$ and the attention distribution $p(z \mid x, q)$ is now modeled as a *conditional random field*, specifying the structure of the z variables. We also assume now that the annotation function f factors into clique annotation functions $f(x,z) = \sum_C f_C(x, z_C)$, where the summation is over the C factors, ψ_C, of the CRF. Our context vector takes the form:

$$c = \mathbb{E}_{z \sim p(z|x,q)}[f(x,z)] = \sum_C \mathbb{E}_{z \sim p(z_C|x,q)}[f_C(x, z_C)] \tag{527}$$

$$p(z \mid x, q) = \frac{1}{Z(x,q)} \prod_C \psi_C(z_C) \tag{528}$$

[122] Also called the "alignments". It is the output of the softmax layer of attention scores in the majority of cases.

[123] In all applications I've seen, $f(x,z) = x_z$.

Example 1: Subsequence Selection. Let $m = T$, and let each $z_i \in \{0, 1\}$ be a binary R.V. Let $f(x, z) = \sum_t^T f_t(x, z_t) = \sum_t^T \boldsymbol{x}_t \mathbb{1}_{z_t=1}$[124]. This yields the context vector,

$$c = \mathbb{E}_{z_1,\ldots,z_T}\left[f(x, z)\right] = \sum_t^T p(z_t = 1 \mid x, q)\boldsymbol{x}_t \tag{529}$$

Although this looks similar to equation 525, we haven't yet revealed the functional form for $p(z \mid x, q)$. Two possible choices:

Linear-Chain CRF:
$$p(z_1, \ldots, z_T \mid x, q) = \frac{1}{Z(x, q)} \prod_t^T \psi_t(z_t, z_{t-1}) \tag{530}$$

The factor ψ for the CRF is **NOT** the same as the factor for the Bernoulli!

Bernoulli:
$$p(z_1, \ldots, z_T \mid x, q) = \prod_t^T p(z_t = 1 \mid x, q) = \prod_t^T \sigma(\psi_t(z_t)) \tag{531}$$

These show why equation 529 is fundamentally different than equation 525:

- It allows for multiple inputs (or no inputs) to be selected for a given query.
- We can incorporate structural dependencies across the z_t's.

Also note that all methods can use potentials from the same neural network or RNN that takes x and q as input. By this we mean, for example, that we can take the same parameters we'd use when computing the scores in our attention layer, and reinterpret them as e.g. CRF parameters. Then, we can compute the marginals $p(z_t \mid x)$ using the forward-backward algorithm[125].

> *Crucially this generalization from vector softmax to forward-backward is just a* ***series of differentiable steps****, and we can compute gradients of its output (marginals) with respect to its input (potentials), allowing the structured attention model to be trained end-to-end as part of a deep model.*

[124] Ok, so equivalently, $z^T x$, i.e. the indicator function can just be replace by z_t here

[125] This is different than the simple softmax we usually use in an attention layer, which does not model any interdependencies between the z_t. The marginals we end up with when using the CRF originate from a *joint* distribution over the entire sequence $z_1, \ldots z_T$. This seems potentially incredibly powerful. Need to analyze in depth.

Do and Artieres, "Neural Conditional Random Fields," (2010).

Neural CRFs. Essentially, we feed the input sequence $\boldsymbol{x}$ through a feed-forward network whose output layer has a linear activation function. The output layer is then connected with the target variable sequence $\mathbf{Y}$. In other words, instead of feeding instances $\boldsymbol{x}$ of the observation variables $\mathbf{X}$, we feed the hidden layer activations of the NN. This results in the conditional probability

$$p(\boldsymbol{y} \mid \boldsymbol{x}) \propto \prod_{c \in C} e^{-E_c(\boldsymbol{x}, \boldsymbol{y}_c, \boldsymbol{w})} = \prod_{c \in C} e^{\langle \boldsymbol{w}_c^{\boldsymbol{y}_c}, \boldsymbol{\Phi}_c(\boldsymbol{x}, \boldsymbol{w}_{NN}) \rangle} \tag{532}$$

We can set $\Phi_c = \Phi$ for a shared-weights approach

where

- $\boldsymbol{w}_{NN}$ are the weights for the NN.
- $\boldsymbol{w}_c^{\boldsymbol{y}_c}$ are the weights (for clique c) for the CRF.
- $\boldsymbol{\Phi}_c(\boldsymbol{x}, \boldsymbol{w}_{NN})$ is the output of the NN. It symbolizes the high-level feature representation of the input $\boldsymbol{x}$ at clique c computed by the NN.

The authors refer to the linear output layer (containing the CRF weights) as the *energy outputs*. For the sake of writing this in more familiar notation for the linear-chain CRF case, here is the above equation translated for the case where each clique corresponds to a timestep t of the input sequence and is either a label-label clique or a state-label clique.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(x)} \prod_t^T \exp\left\{-E_t(\boldsymbol{x}, \boldsymbol{y}_t, \boldsymbol{y}_{t-1}, \boldsymbol{w})\right\} \tag{533}$$

$$= \frac{1}{Z(x)} \prod_t^T \exp\left\{-E_{loc}(\boldsymbol{x}, t, y_t, \boldsymbol{w}) - E_{trans}(\boldsymbol{x}, t, y_{t-1}, y_t, \boldsymbol{w})\right\} \tag{534}$$

$$\tag{535}$$

where the authors are using a blanket $\boldsymbol{w}$ to denote all model parameters[126].

[126] Also note that the authors allow for utilizing the input sequence $\boldsymbol{x}$ in the transition energy function, E_{trans}, although usually we implement E_{trans} using only y_{t-1} and y_t.

Initialization & Fine-Tuning. The hidden layers of the NN are initialized layer-by-layer in an unsupervised manner using RBMs. It's important to note that the hidden layers of the NN consist of binary units. Then, using the pre-trained hidden layers, the CRF layer is initialized by training it in the usual way, and keeping the pretrained NN weights fixed.

Next, fine-tuning is used to learn all parameters globally.

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}} \tag{536}$$

$$\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{E}(\boldsymbol{x}^{(i)})}\frac{\partial \boldsymbol{E}(\boldsymbol{x}^{(i)})}{\partial \boldsymbol{w}} \tag{537}$$

$$\left[\boldsymbol{E}(\boldsymbol{x}^{(i)})\right]_t = E_{loc}(\boldsymbol{x}, t, y_t, \boldsymbol{w}) + E_{trans}(\boldsymbol{x}, t, y_{t-1}, y_t, \boldsymbol{w}) \tag{538}$$

$$\tag{539}$$

where $\frac{\partial \boldsymbol{E}_i}{\partial \boldsymbol{w}}$ is the Jacobian matrix of the NN outputs for input sequence $\boldsymbol{x}^{(i)}$ w.r.t. weights $\boldsymbol{w}$. By setting $\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{E}_i}$ as backprop errors of the NN output units, we can backpropagate and get $\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}}$ using the chain rule over the hidden layers.

Bidirectional LSTM-CRF Models for Sequence Tagging

 Written by Brandon McKinzie

Huang et al., "Bidirectional LSTM-CRF Models for Sequence Tagging," (2015).

BI-LSTM-CRF Networks. Consider the matrix of scores $f_\theta(\boldsymbol{x})$ for input sentence $\boldsymbol{x}$. The element $[f_\theta]_{\ell,t}$ gives the score for label ℓ with the t-th word. This is output by the LSTM network parameterized by θ. We let $[A]_{i,j}$ denote the transition score from label i to label j within the CRF. The total set of parameters is denoted $\widetilde{\theta} = \theta \cup \boldsymbol{A}$. The total score for input sentence $\boldsymbol{x}$ and predicted label sequence y is then

$$s(\boldsymbol{x}, \boldsymbol{y}, \widetilde{\theta}) = \sum_t^T \left(A_{y_{t-1}, y_t} + [f_\theta]_{y_t, t} \right) \tag{540}$$

Features. The authors incorporate 3 distinct types of input features:

- **Spelling features**. Various standard lexical/syntactical features. One-hot encoded as usual.
- **Context features**. Unigram, bigram, and sometimes tri-gram features. One-hot encoded.
- **Word embeddings**. Distinct from the word features. Use a pretrained embedding for each word.

Although it's not entirely clear, it appears they concatenate all of the aforementioned features together as input to the BI-LSTM. This necessarily means they are learning an embedding for the one-hot encoded spelling and word features. They also add direct connections from the input to the CRF for the spelling and word features.

EDIT: they may actually replace the one-hot encoded word features with the word embeddings. Unclear.

Pawar et al., "Relation Extraction: A Survey," (December 2017).

Feature-based Methods. For each entity pair (e_1, e_2), generate a set of features and train a classifier to predict the relation[127]. Some useful features are shown in the figure below.

Feature Types	Example
Words: Words of both the mentions and all the words in between	M11_leaders, M21_Venice; B1_of, B2_Italy, B3_'s, B4_left-wing, B5_government, B6_were, B7_in
Entity Types: Entity types of both the mentions	E1_PERSON, E2_GPE
Mention Level: Mention types (NAME, NOMINAL or PRONOUN) of both the mentions	M1_NOMINAL, M2_NAME
Overlap: #words separating the two mentions, #other mentions in between, flags indicating whether the two mentions are in the same NP, VP or PP	7_Words_Apart, 2_Mentions_In_Between (Italy & government), Not_Same_NP, Not_Same_VP, Not_Same_PP
Dependency: Words, POS and chunk labels of words on which the mentions are dependent in the dependency tree, #links traversed in dependency tree to go from one mentions to another	M1W_were, M1P_VBD, M1C_VP, M2W_in, M2P_IN, M2C_PP, DepLinks_3
Parse Tree: Path of non-terminals connecting the two mentions in the parse tree, and the path annotated with head words	PERSON-NP-S-VP-PP-GPE, PERSON-NP:leaders-S -VP:were-PP:in-GPE

Authors found that SVMs outperform MaxEnt (logistic reg) classifiers for this task.

Kernel methods. Instead of explicit feature engineering, we can design kernel functions for computing similarities between representations of two relation instances[128] (a relation instance is a triplet of the form (e_1, e_2)), and SVM for the classification.

[127] If there are N unique relations for our data, it is common to train the classifier to predict $2N$ total relations, to handle both possible orderings of relation arguments.

[128] Recall that kernel methods are for the general optimization problem

$$\min_w \sum_{i=1}^n \text{loss}\left(\sum_{j=1}^n \alpha_j \phi(x_j)^T \phi(x_i), y_i\right) + \lambda \sum_{j=1}^n \sum_{k=1}^n \alpha_j \alpha_k \phi(x_j)^T \phi(x_k) \tag{541}$$

which changes the direct focus from feature engineering to "similarity" engineering.

One approach is the **sequence kernel**. We represent each relation instance as a sequence of feature vectors:

$$(e_1, e_2) \to (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_N)$$

where N might be e.g. the number of words between the two entities, and the dimension of each f is the same, and could correspond to e.g. POS tag, NER tag, etc. More formally, define the *generalized subsequence kernel*, $K_n(s, t, \lambda)$, that computes some number of weighted subsequences u such that

- There exist index sequences $ii := (i_1, \ldots i_n)$ and $jj := (j_1, \ldots j_n)$ of length n such that

$$u_i \in \boldsymbol{s}_i \qquad \forall i \in ii \tag{542}$$

$$u_j \in \boldsymbol{t}_j \qquad \forall j \in jj \tag{543}$$

$$\tag{544}$$

- The weight of u is $\lambda^{l(ii)+l(jj)}$, where $l(x) = \max(x) - \min(x)$ and $0 < \lambda < 1$. Sparser (more spaced out) subsequences get lower weight.

The authors then provide the recursion formulas for K, and describe some extensions of sequence kernels for relation extraction.

Syntactic Tree Kernels. Structural properties of a sentence are encoded by its constituent parse tree. The tree defines the syntax of the sentence in terms of constituents such as noun phrases (NP), verb phrases (VP), prepositional phrases (PP), POS tags (NN, VB, IN, etc.) as non-terminals and actual words as leaves. The syntax is usually governed by Context Free Grammar (CFG). Constructing a constituent parse tree for a given sentence is called *parsing*. The **Convolution Parse Tree Kernel** K_T can be used for computing similarity between two syntactic trees.

Syntactic Tree Kernels

Dependency Tree Kernels. For grammatical relations between words in a sentence. Words are the nodes and dependency relations are the edges (in the tree), typically from dependent to parent. In the **relation instance representation**, we use the smallest subtree containing the entity pair of a given sentence. Each node is augmented with additional features like POS, chunk, entity level (name, nominal, pronoun), hypernyms, relation argument, etc. Formally, an *augmented dependency tree* is defined as a tree T where

Dependency Tree Kernels

- Each node t_i has features $\phi(t_i) = \{v_1, \ldots, v_d\}$.
- Let $t_i[\boldsymbol{c}]$ denote all children of t_i, and let $Pa(t_i)$ denote its parent.
- For comparison of two nodes we use:
  - **Matching function** $m(t_i, t_j)$: equal to 1 if some important features are shared between t_i and t_j, else 0.
  - **Similarity function** $s(t_i, t_j)$: returns a positive real similarity score, and defined as

$$s(t_i, t_j) = \sum_{v_q \in \phi(t_i)} \sum_{v_r \in \phi(t_j)} \text{Compat}(v_q, v_r) \tag{545}$$

  over some compatibility function between two feature values.

Finally, we can define the overall dependency tree kernel $K(T_1, T_2)$ for similarity between trees T_1 and T_2 as follows. Let r_i denote the root node of tree T_i.

$$K(T_1, T_2) = \begin{cases} 0 & \text{if } m\,(r_1, r_2) = 0 \\ s(r_1, r_2) + K_c(r_1[\boldsymbol{c}], r_2[\boldsymbol{c}]) & \text{otherwise} \end{cases} \tag{546}$$

$$K_c(t_i[\boldsymbol{c}], t_j[\boldsymbol{c}]) = \sum_{\boldsymbol{a},\boldsymbol{b},l(\boldsymbol{a})=l(\boldsymbol{b})} \lambda^{d(\boldsymbol{a})+d(\boldsymbol{b})} K(t_i[\boldsymbol{a}], t_j[\boldsymbol{b}]) \tag{547}$$

$a_1 \leq a_2 \leq \ldots \leq a_n$

$d(\boldsymbol{a}) \triangleq a_n - a_1 + 1$

$0 < \lambda < 1$

The interpretation is that, whenever a pair of matching nodes is found, *all* possible matching subsequences[129] of their children are found. Two subsequences $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to "match" if $m(a_i, b_1) = 1 (\forall i < n)$. Similar to the sequence kernel seen earlier, λ is a decay factor that penalizes sparser subsequences.

[129]Note that a summation over subsequences of a sequence $\boldsymbol{a}$, denoted here as $\sum_{\boldsymbol{a}}$, expands to $\{a_1, \ldots a_n, a_1a_2, a_1a_3, \ldots a_1a_n, a_1a_2a_3, \ldots, a_2a_5a_6, \ldots\}$ and so on and so forth.

Neural Relation Extraction with Selective Attention over Instances

Written by Brandon McKinzie

Lin et al., "Neural Relation Extraction with Selective Attention over Instances," (2016).

Introduction. A common distant supervision approach for RE is aligning a KB with text. For any (e_1, r, e_2) in the KB, it assumes that all text mentions of (e_1, e_2) express the relation r. Of course, this assumption will often not be true. This motivates the notion of **multi-instance learning**, wherein we predict whether a set of instances $\{x_1, \dots x_n\}$ (each of which contain mention(s) of (e_1, e_2)) imply the existence of (e_1, r, e_2) being true.

Input Representation. Each instance sentence x is tokenized into a sequence of words. Each word w_i is transformed into a concatenation $\boldsymbol{w}_i \in \mathbb{R}^d$ $(d = d_w + 2d_{pos})$,

$$\boldsymbol{w}_i := [\text{word2Vec}(w_i); \text{dist}(w_i, e_1); \text{dist}(w_i, e_2)] \tag{548}$$

where dist(a, b) returns the [embedded] relative distance (num tokens) between a and b in the given sentence (positive integer)[130].

Convolutional Network. We use a CNN to encode a sentence of embeddings $\{\boldsymbol{w}_1, \dots, \boldsymbol{w}_T\}$ into a single sentence vector representation $\boldsymbol{x}$. Denote the kernel/filter/window size as ℓ and the number of words in the given sentence as T. Let $\boldsymbol{q}_i \in \mathbb{R}^{\ell \cdot d}$ denote the vector for the ith window,

$$\boldsymbol{q}_i = \boldsymbol{w}_{i-\ell+1:i} \qquad (1 \leq i \leq T + \ell - 1) \tag{549}$$

and let $\boldsymbol{Q} \in \mathbb{R}^{(T+\ell-1)\times \ell \cdot d}$ be defined such that row $\boldsymbol{Q}_i = \boldsymbol{q}_i^T$. It follows that, for convolution matrix $\boldsymbol{W} \in \mathbb{R}^{K \times (\ell \cdot d)}$, the output of the kth filter, and subsequent max-pooling, is

$$\boldsymbol{p}_k = \left[\boldsymbol{W}\boldsymbol{Q}^T\right]_k + \boldsymbol{b} \tag{550}$$

$$[\boldsymbol{x}]_k = [\max(\boldsymbol{p}_{k1}); \max(\boldsymbol{p}_{k2}); \max(\boldsymbol{p}_{k3})] \tag{551}$$

where we've divided $\boldsymbol{p}_k$ into three segments, corresponding to before entity 1, middle, and after entity 2 of the given sentence. The sentence vector $\boldsymbol{x} \in \mathbb{R}^{3K}$ is the concatenation of all of these, after feeding through a non-linear activation function like a ReLU.

[130] More specifically, it is an embedded representation of the relative distance. To actually implement it, you'd first shift the relative distances such that they begin at 0, and learn 2 * window_size + 1 embedding vectors for each of the possible position offsets. Anything outside the window is embedded into the zero vector.

Selective Attention over Instances. An attention mechanism is employed over all n sentence instances x_i for some candidate entity pair (e_1, e_2). The output is a **set vector** $\boldsymbol{s}$, a real-valued vector representation of the set of instances, where

$$\boldsymbol{s} = \sum_i \alpha_i \boldsymbol{x}_i \tag{552}$$

$$\alpha_i = \text{softmax}(\boldsymbol{x}_i \boldsymbol{A} \boldsymbol{r}) \tag{553}$$

is an attention-weighted sum over the instance embeddings. Note that they constrain $\boldsymbol{A}$ to be diagonal. Finally, the predictive distribution is defined as

$$p(r \mid \mathcal{S}, (e_1, e_2)\theta) = \text{softmax}\left(\boldsymbol{M}\boldsymbol{s} + \boldsymbol{d}\right)_r \tag{554}$$

where $\mathcal{S}$ is the set of n sentences for the given entity pair (e_1, e_2).

On Herding and the Perceptron Cycling Theorem

Written by Brandon McKinzie

Gelfand et al., "On Herding and the Perceptron Cycling Theorem," (2010).

Introduction. Begin with the familiar learning rule of Rosenblatt's perceptron, after some *incorrect* prediction $\hat{y}_i = \text{sgn}(\boldsymbol{w}^T \boldsymbol{x}_i)$,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{x}_i(y_i - \hat{y}_i) \tag{555}$$

which has the effect that a subsequent prediction on $\boldsymbol{x}_i$ will (before taking the sign) be $||\boldsymbol{x}_i||_2^2$ closer to the correct side of the hyperplane. The **perceptron cycling theorem** (PCT) states that if the data is *not* linearly separable, the weights will still remain bounded and won't diverge to infinity. **This paper shows that the PCT implies that certain moments are conserved on average**. Formally, their result says that, for some N number of iterations over samples selected from training data (with replacement)[131],

$$\left|\left|\frac{1}{N}\sum_i^N \boldsymbol{x}_i y_i - \frac{1}{N}\sum_i^N \boldsymbol{x}_i \hat{y}_i\right|\right| \sim \mathcal{O}\left(\frac{1}{N}\right) \tag{556}$$

where it's important to remember that $\hat{y}_i$ here is the prediction for $\boldsymbol{x}_i$ when it was encountered at that training iteration. This result shows that perceptron learning generate predictions that correlate with the input attributes the same way as the true labels do, and [the correlations] converge to the sample mean with a rate of 1/N. This also hints at why averaged perceptron algorithms (using the average of weights across training) makes sense, as opposed to just selected the best weights. This paper also shows that *supervised perceptron algorithms and unsupervised herding algorithms can all be derived from the PCT.*

Below are some theorems that will be used throughout the paper. Let $\{\boldsymbol{w}_t\}$ be a sequence of vectors $\boldsymbol{w}_t \in \mathbb{R}^D$, each generated according to iterative updates $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \boldsymbol{v}_t$, where $\boldsymbol{v}_t$ is an element of a *finite* set $\boldsymbol{V}$, and the norm of $\boldsymbol{v}_t$ is bounded: $\max ||\boldsymbol{v}_t|| = R < \infty$.

> ***PCT***: *$\forall t \geq 0$: If $\boldsymbol{w}_t^T \boldsymbol{v}_t \leq 0$, $\exists M > 0$ s.t. $||\boldsymbol{w}_t - \boldsymbol{w}_0|| < M$.*
> ***Convergence Thm***: *If PCT holds, then $||\frac{1}{T}\sum_{t=1}^T \boldsymbol{v}_t|| \sim \mathcal{O}\frac{1}{T}$.*

[131] Unclear whether this is only for samples that corresponded to an update, or just all samples during training.

Herding. Consider a fully observed Markov Random Field (MRF) over m variables, each of which can take on an integer value in the range $[1, K]$. In herding, our energy function and weight updates for observation $\boldsymbol{x}$ (over all m variables in $\mathcal{X}$),

$$E(\boldsymbol{x}) = -\boldsymbol{w}^T \phi(\boldsymbol{x}) \tag{557}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \bar{\phi} - \phi(\boldsymbol{x}_t^*) \tag{558}$$

$$\text{where} \qquad \bar{\phi} = \mathbb{E}_{\boldsymbol{x}^{(i)} \sim p_{data}} \left[\phi(\boldsymbol{x}^{(i)}) \right] \tag{559}$$

$$\text{and} \qquad \boldsymbol{x}_t^* = \arg\max_{\boldsymbol{x}} \boldsymbol{w}_t^T \phi(\boldsymbol{x}) \tag{560}$$

What if we consider more complicated features that depend on the weights $\boldsymbol{w}$? This situation may arise in e.g. models with hidden units $\boldsymbol{z}$, where our feature function would take the form $\phi(\boldsymbol{x}, \boldsymbol{z})$, and we always select $\boldsymbol{z}$ via

$$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{w}) = \arg\max_{\boldsymbol{z}'} \boldsymbol{w}^T \phi(\boldsymbol{x}, \boldsymbol{z}') \tag{561}$$

and therefore our feature function ϕ depends on weights $\boldsymbol{w}$ through $\boldsymbol{z}$. In this case, our herding update terms from above take the form

$$\bar{\phi} = \mathbb{E}_{\boldsymbol{x}^{(i)} \sim p_{data}} \left[\phi(\boldsymbol{x}^{(i)}, \boldsymbol{z}(\boldsymbol{x}^{(i)}, \boldsymbol{w})) \right] \tag{562}$$

$$\boldsymbol{x}_t^*, \boldsymbol{z}_t^* = \arg\max_{\boldsymbol{x}, \boldsymbol{z}} \boldsymbol{w}_t^T \phi(\boldsymbol{x}, \boldsymbol{z}) \tag{563}$$

Conditional Herding. Main contribution of this paper. It's basically identical to regular herding, but now we decompose $\boldsymbol{x}$ into inputs and outputs $(\boldsymbol{x}, \boldsymbol{y})$ for interpreting in a discriminative setting. In the paper, they express $\boldsymbol{w}^T \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ identically as a discriminative RBM. The parameter update for mini-batch $\mathcal{D}_t$ is given by

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \frac{\boldsymbol{\eta}}{|\mathcal{D}_t|} \sum_{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \in \mathcal{D}_t} \left(\phi(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{z}) - \phi(\boldsymbol{x}^{(i)}, \boldsymbol{y}^*, \boldsymbol{z}^*) \right) \tag{564}$$

Non-Convex Optimization for Machine Learning

P. Jain and P. Kar, "Non-convex Optimization for Machine Learning," (2017).

Convex Analysis (2.1). First, let's summarize some definitions.

Convex Combination

*A **convex combination** of a set of n vectors $\boldsymbol{x}_i \in \mathbb{R}^p$, $i = 1 \dots n$ is a vector $\boldsymbol{x}_\theta := \sum_{i=1}^n \theta_i \boldsymbol{x}_i$, where $\theta_i \geq 0$ and $\sum_{i=1}^n \theta_i = 1$.*
My interp: A weighted average where the weights can be interpreted as probability mass associated with each vector.

Convex Set. Sets that contain all [points in] line segments that join any 2 points in the set.

*A set $\mathcal{C}$ is called a **convex set** if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{C}$ and $\lambda \in [0, 1]$, we have that $(1-\lambda)\boldsymbol{x} + \lambda \boldsymbol{y} \in \mathcal{C}$ as well.*

Proving conv. comb. of 3 vectors $\in \mathcal{C}$ too.

After reading the definition of a convex set above, it seemed intuitive that any convex combination of points $\in \mathcal{C}$ should also be in it as well (i.e. generalizing the pairwise definition). Let $x, y, z \in \mathcal{C}$. How can we prove that $\theta_1 x + \theta_2 y + \theta_3 z$ (where θ_i satisfy the constraints of a convex comb.) is also in $\mathcal{C}$? Here is how I ended up doing it:

- If we can prove that $\theta_1 x + \theta_2 y = (1 - \theta_3)v$ for some $v \in \mathcal{C}$, then our work is done. This is pretty easy to show via simple arithmetic.
- Case 1: assume $\theta_3 < 1$, so that we can divide both sides by $1 - \theta_3$:

$$v = \frac{\theta_1}{1-\theta_3} x + \frac{\theta_2}{1-\theta_3}$$

  Clearly, the two coefficients here sum 1 and satisfy the constraints of a convex combination, and therefore we know that $v \in \mathcal{C}$, and this case is done.
- Case 2: assume $\theta_3 = 1$. Well, that means $\theta_1 = \theta_2 = 0$. Trivially, $z \in \mathcal{C}$ and this case is done.

Convex Function

*A continuously differentiable function $f : \mathbb{R}^p \mapsto \mathbb{R}$ is a **convex function** if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, we have that*

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \tag{565}$$

While thinking about how to gain intuition for the above, I came across chapter 3 of "Convex Optimization" which describes this in much more detail. It's crucial to recognize that the RHS of the inequality is the 1st-order Taylor expansion of the function f about $\boldsymbol{x}$, evaluated at $\boldsymbol{y}$. In other words, *the first-order Taylor approximation is a **global underestimator** of any convex function f*[132].

[132] Consider what this implies about all the 1st-order gradient-based optimizers we use.

Strongly Convex/Smooth Function. Informally, strong convexity ensures a convex function doesn't grow too *slow*, while strong smoothness ensures a ~~convex~~[133] function doesn't grow too *fast*. Formally,

A continuously differentiable function is considered ***α-strongly convex*** *(SC) and* ***β-strongly smooth*** *(SS) if* $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$ *we have*

$$\frac{\alpha}{2}||\boldsymbol{x}-\boldsymbol{y}||_2^2 \leq f(\boldsymbol{y}) - f(\boldsymbol{x}) - \langle \nabla f(\boldsymbol{x}), \boldsymbol{y}-\boldsymbol{x} \rangle \leq \frac{\beta}{2}||\boldsymbol{x}-\boldsymbol{y}||_2^2 \tag{566}$$

Considering the aforementioned 1st-order Taylor approximation interpretation, we see that α determines just how much larger $f(\boldsymbol{y})$ must be compared to its linear approximation. Conversely, β determines the upper bound for how large this discrepancy is allowed to be[134].

Exercise 2.1: SS does not imply convexity

Construct a ***non-convex*** *function* $f : \mathbb{R}^p \mapsto \mathbb{R}$ *that is* 1*-SS.*

We need to find a function whose linear approximation is always more than $\frac{1}{2}$ times the magnitude of the difference in inputs **squared**, compared to the true value. Intuitively, I'd expect any *concave* function to satisfy this, since its linear approximation is a global *overestimator* of the true value. So, for example, $f(\boldsymbol{y}) = -||\boldsymbol{y}||_2^2$ would satisfy $1-SS$ while being non-convex.

Lipschitz Function

A function f *is* ***B-Lipschitz*** *if* $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$,

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq B \cdot ||\boldsymbol{x}-\boldsymbol{y}||_2 \tag{567}$$

Jensen's Inequality. Generalizes behavior of convex functions on convex combinations[135].

If X *is a R.V. taking values in the domain of a convex function* f*, then*

$$\mathbb{E}\left[f(X)\right] \geq f(\mathbb{E}\left[X\right]) \tag{568}$$

[133] Strong smoothness alone does not imply convexity.

[134] Notice that SC and SS are *quadratic* lower and upper bounds, respectively. This means that the allowed deltas grow as a function of the distance between $\boldsymbol{x}$ and $\boldsymbol{y}$, whereas things like Lipschitzness grow linearly.

[135] It should be obvious that expectations are convex combinations.

Convex Projections (2.2). Given any closed set $\mathcal{C} \in \mathbb{R}^p$, the projection operator $\Pi_{\mathcal{C}}(\cdot)$ is defined as

$$\Pi_{\mathcal{C}}(\boldsymbol{z}) := \arg\min_{\boldsymbol{x} \in \mathcal{C}} ||\boldsymbol{x} - \boldsymbol{z}||_2 \tag{569}$$

If $\mathcal{C}$ is a convex set, then the above reduces to a convex optimization problem. Projections onto convex sets have three particularly interesting properties. For each of them, the setup is: "For any convex set $\mathcal{C} \subset \mathbb{R}^p$, and any $\boldsymbol{z} \in \mathbb{R}^p$, let $\hat{\boldsymbol{z}} := \Pi_{\mathcal{C}}(\boldsymbol{z})$. Then $\forall \boldsymbol{x} \in \mathcal{C}, \ldots$"

- **Property-O**[136]: $||\hat{\boldsymbol{z}} - \boldsymbol{z}||_2 \leq ||\boldsymbol{x} - \boldsymbol{z}||_2$. Informally: "the projection results in the point $\hat{\boldsymbol{z}}$ in $\mathcal{C}$ that is closest to the original $\boldsymbol{z}$". This basically just restates the optimization problem.
- **Property-I**. $\langle \boldsymbol{x} - \hat{\boldsymbol{z}}, \boldsymbol{z} - \hat{\boldsymbol{z}} \rangle \leq 0$. Informally: "from the perspective of $\hat{\boldsymbol{z}}$, all points $\boldsymbol{x} \in \mathcal{C}$ are in the (informally) opposite direction of $\boldsymbol{z}$."
- **Property-II**. $||\hat{\boldsymbol{z}} - \boldsymbol{x}||_2 \leq ||\boldsymbol{z} - \boldsymbol{x}||_2$. Informally: "the projection brings the point closer to all points in $\mathcal{C}$ compared to its original location."

Proving Property-I

A proof by contradiction.

1. Assume that $\exists \boldsymbol{x} \in \mathcal{C}$ s.t. $\langle \boldsymbol{x} - \hat{\boldsymbol{z}}, \boldsymbol{z} - \hat{\boldsymbol{z}} \rangle > 0$.
2. We know that $\hat{\boldsymbol{z}}$ is also in $\mathcal{C}$, and since $\mathcal{C}$ is convex, then for any $\lambda \in [0, 1]$,

$$\boldsymbol{x}_\lambda := \lambda x + (1 - \lambda)\hat{\boldsymbol{z}} \tag{570}$$

   must also be in $\mathcal{C}$.
3. If we can show that some value of λ guarantees that $||\boldsymbol{z} - \boldsymbol{x}_\lambda||_2 < ||\boldsymbol{z} - \hat{\boldsymbol{z}}||_2$, this would directly contradict property-O, implying $\hat{\boldsymbol{z}}$ is not the closest member of $\mathcal{C}$ to $\boldsymbol{z}$. I'm not sure how to actually derive the range of λ values that satisfy this, though.

(Convex) Projected Gradient Descent (2.3). Our optimization problem is

$$\min_{\boldsymbol{x} \in \mathbb{R}^p} f(\boldsymbol{x}) \quad \text{s.t.} \quad \boldsymbol{x} \in \mathcal{C} \tag{571}$$

where $\mathcal{C} \subset \mathbb{R}^p$ is a convex constraint set, and $f : \mathbb{R}^p \mapsto \mathbb{R}$ is a convex objective function. Projected gradient descent iteratively updates the value of $\boldsymbol{x}$ that minimizes f as usual, but additionally projects the current iterate (value of best $\boldsymbol{x}$) onto $\mathcal{C}$ at the end of each iteration. That's the only difference.

[136] In this case only, $\mathcal{C}$ need not be convex

Non-Convex Projections (3.1). Here we look at a few special cases where projecting onto a non-convex set can still be carried out efficiently.

- **Projecting into sparse vectors**. The set of s-sparse vectors (vectors with at most s nonzero elements) is denoted as

$$\mathcal{B}_0(s) \triangleq \{\boldsymbol{x} \in \mathbb{R}^p \mid ||\boldsymbol{x}||_0 \leq s\} \tag{572}$$

  It turns out that $\hat{\boldsymbol{z}} := \Pi_{\mathcal{B}_0(s)}(\boldsymbol{z})$ is obtained by setting all except the top-s elements of $\boldsymbol{z}$ to zero.
- **Projecting into low-rank matrices**. The set of $m \times n$ matrices with rank at most r is denoted as

$$\mathcal{B}_{rank}(r) \triangleq \{A \in \mathbb{R}^{m \times n} \mid \text{rank}(A) \leq r\} \tag{573}$$

  and we want to project some matrix A onto this set,

$$\Pi_{\mathcal{B}_{rank}(r)}(A) := \underset{X \in \mathcal{B}_{rank}(r)}{\arg\min} ||A - X||_F \tag{574}$$

  This can be done efficiently via SVD on A and retaining the top r singular values and vectors.

Restricted Strong Convexity and Smoothness (3.2).

Restricted Convexity

A continuously differentiable function $f : \mathbb{R}^p \mapsto \mathbb{R}$ is said to satisfy restricted convexity over a (possibly non-convex) region $\mathcal{C} \subseteq \mathbb{R}^p$ if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{C}$, we have that

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \tag{575}$$

and a similar rephrasing for restricted strong convexity (RSC) and restricted strong smoothness (RSS).

Improving Language Understanding by Generative Pre-Training

Written by Brandon McKinzie

Radford et al., "Improving Language Understanding by Generative Pre-Training," (2018).

Unsupervised Pre-Training (3.1). Given unsupervised corpus of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$, train with a standard LM objective:

$$L_1(\mathcal{U}) = \sum_i^n \log P(u_i \mid u_{i-k}, \ldots, u_{i-1}; \Theta) \tag{576}$$

The authors use a **Transformer decoder**, i.e. literally just the decoder part of the Transformer in "Attention is all you need."

Supervised Fine-Tuning (3.2). Now we have a labeled corpus $\mathcal{C}$, where each instance consists of a sequence of input tokens $x^1, \ldots, x^m$, along with a label y. They just feed the inputs through the transformer until they obtain the final transformer block's activation h_l^m, and linearly project it to output space:

$$P(y \mid x^1, \ldots, x^m) = \text{softmax}(h_l^m W_y) \tag{577}$$

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y \mid x^1, \ldots, x^m) \tag{578}$$

They also found that including a language modeling auxiliary objective helped learning,

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda L_1(\mathcal{C}) \tag{579}$$

...that's it. Extremely simple, yet somehow effective.

Deep Contextualized Word Representations

Written by Brandon McKinzie

Peters et al., "Deep Contextualized Word Representations," (2018).

Bidirectional Language Models (3.1). Given a sequence of N tokens, a forward LM computes the probability of the sequence via

$$p(t_1, \ldots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, \ldots, t_{k-1}) \tag{580}$$

A common approach is learning context-independent token representations $\boldsymbol{x}_k$ and passing these through L layers of forward LSTMs. The top layer LSTM output at step k, $\overrightarrow{\boldsymbol{h}}_{k,L}$, is used to predict t_{k+1} with a softmax layer. The authors' biLM combines a forward and backward LM to jointly maximize

$$\begin{aligned}\sum_{k=1}^{N} \Big[&\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) \\ &+ \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big]\end{aligned} \tag{581}$$

and it's important to note the shared parameters Θ_x (token representation) and Θ_s (output softmax).

ELMo (3.2). A task-specific linear combination of the intermediate representations.

$$\boldsymbol{ELMo}_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \boldsymbol{h}_{k,j}^{LM} \tag{582}$$

where $\boldsymbol{s}^{task}$ are softmax-normalized weights (so the combination is convex). The authors also mention that, in some cases, it helped to apply **layer normalization** to each biLM layer before weighting.

Using biLMs for Supervised NLP (3.3). Given a pretrained biLM and a supervised architecture, we can learn the ELMo representations (jointly with the given supervised task) as follows.

1. Freeze the weights of the [pretrained] biLM.
2. Concatenate the token representations (e.g. GloVe) with the ELMo representation.
3. Pass the concatenated representation into the supervised architecture.

The authors found it beneficial to some dropout to ELMo, and in some cases add L2-regularization on the ELMo weights.

Pretrained biLM Architecture (3.4). In addition to the biLM we introduced earlier, the authors make the following changes/specifications for their pretrained biLMs:

- **residual connections** between LSTM layers[137].
- Halved all embedding and hidden dimensions from the CNN-BIG-LSTM model in *Exploring the Limits of Language Modeling*.
- The $\boldsymbol{x}_k$ token representations use 2048 character n-gram convolutional filters followed by two **highway layers**.

[137] So the output of some layer, instead of being LSTM(x), becomes (x + LSTM(x))

Josefina et al., "Exploring the Limits of Language Modeling," (2016).

NCE and Importance Sampling (3.1). In this section, assume any $p(w)$ is shorthand for $p(w \mid \{w_{prev}\})$.

- **Noise Contrastive Estimation** (NCE). Train a classifier to discriminate between true data (from distribution p_d) or samples coming from some arbitrary noise distribution p_n. If these distributions were known, we could compute

$$p(Y{=}true \mid w) = \frac{p(w \mid \text{true})p(\text{true})}{p(w)} \tag{583}$$

$$= \frac{p_d(w)p(\text{true})}{p(w, \text{true}) + p(w, \text{false})} \tag{584}$$

$$= \frac{p_d(w)p(\text{true})}{p_d(w)p(\text{true}) + p_n(w)p(\text{false})} \tag{585}$$

$$= \frac{p_d(w)}{p_d(w) + kp_n(w)} \tag{586}$$

  where k is the number of negative samples per positive word. The idea is to train a logistic classifier $p(Y{=}true \mid w) = \sigma(\log p_{model} - \log kp_n(w))$, then $\text{softmax}(\log p_{model})$ is a good approx of $p_d(\boldsymbol{w})$.
- **Importance Sampling**. Estimates the partition function. Consider that now we have a set of $k+1$ words $W = \{w_1, \ldots, w_{k+1}\}$, where w_1 is the word coming from the true data, and the rest are from the noise distribution. We train a multinomial logistic regression over $k+1$ classes,

$$p(Y{=}i \mid W) = \frac{p_d(w_i)}{p_n(w_i)} \frac{1}{\sum_{i'=1}^{k+1} p_d(w_{i'})/p_n(w_{i'})} \tag{587}$$

$$\propto_Y \frac{p_d(w_i)}{p_n(w_i)} \tag{588}$$

  and we end up seeing that IS is the same as NCE, except in the multiclass setting and with cross entropy loss instead of logistic loss.

CNN Softmax (3.2). Typically the logit for word w is given by $z_w = h^T e_w$, where h is often the output state of an LSTM, and e_w is a vector of parameters that could be interpreted as the word embedding for w. Instead of this, the authors propose what they call *CNN Softmax*, where we compute $e_w = CNN(chars_w)$. Although this makes the function mapping from w to e_w much smoother (due to the tied weights), it ends up having a hard time distinguishing between similarly spelled words that may have entirely different meanings. The authors use a correction factor, learned for each word, such that

$$z_w = h^T CNN(chars_w) + h^T M corr_w \tag{589}$$

where M projects low-dimensional $corr_w$ back up to the dimensionality of the LSTM state h.

Char LSTM Predictions (3.3). To reduce the computational burden of the partition function, the authors feed the word-level LSTM state h through a character-level LSTM that predicts the target word one character at a time.

NLP with Deep Learning

Contents

5.1 Word Vector Representations (Lec 2) 201
5.2 GloVe (Lec 3) 204

NLP with Deep Learning **May 08**

Word Vector Representations (Lec 2)

Table of Contents Local *Written by Brandon McKinzie*

Meaning of a word. Common answer is to use a *taxonomy* like WordNet that has hypernyms (is-a) relationships. Problems with this discrete representation: misses nuances, e.g. the words in a set of synonyms can actually have quite different meanings/connotations. Furthermore, viewing words as atomic symbols is a bit like using one-hot vectors of words in a vocabulary space (inefficient).

Distributed representations. Want a way to encode word vectors such that two similar words have a similar structure/representation. The **distributional similarity-based**[138] approach represents words by means of its *neighbors* in the sentences in which it appears. You end up with a dense "vector for each word type, chosen so that it is good at predicting other words appearing in its context."

Skip-gram prediction. Given a word w_t at position t in a sentence, learn to predict [probability of] some number of surrounding words, given w_t. Standard minimization with negative log-likelihood:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m}\log \Pr(w_{t+j}|w_t) \tag{590}$$

$$\Pr(o \mid c) = \frac{e^{\boldsymbol{u}_o^T\boldsymbol{v}_c}}{\sum_{w=1}^{\text{vocab size}} e^{\boldsymbol{u}_w^T\boldsymbol{v}_c}} \tag{591}$$

I cannot believe this actually works.

where

- The params θ are the vector representation of the words (they are the *only* learnable parameters here).
- m is our radius/window size.
- o and c are indices into our vocabulary (somewhat inconsistent notation).
- Yes, they are using different vector representations for $\boldsymbol{u}$ (context words) and $\boldsymbol{v}$ (center words). I'm assuming one reason this is done is because it makes the model architecture simpler/easier to build.

Some subtleties:

[138] Note that this is distinct from the way "distributed" is meant in "distributed representation." In contrast, distributional similarity-based representations refers to the notion that you can describe the meaning of words by understanding the context in which they appear.

- Looks like e.g. $Pr(w_{t+j} \mid w_t)$ doesn't really care what the value of j is, it is just modeling the probability that it is *somewhere* in the context window. The w_t are one-hot vectors into the vocabulary. Standard tricks for simplifying the cross-entropy loss apply.
- Equation 591 should be interpreted as the probability that the oth word in our vocabulary occurs in the context window of the cth word in our vocabulary.
- The model architecture is *identical* to an autoencoder. However, the (big) difference is the training procedure and interpretation of the model parameters going "in" versus the parameters going "out".

Sentence Embeddings. It turns out that simply taking a weighted average of word vectors and doing some PCA/SVD is a competitive way of getting unsupervised word embeddings. Apparently it *beats* supervised learning with LSTMs (?!). The authors claim the theoretical explanation for this method lies in a latent variable generative model for sentences (of course). Approach:

Discussion based on paper by Arora et al., (2017).

1. Compute the weighted average of the word vectors in the sentence:

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{592}$$

The authors call their weighted average the **Smooth Inverse Frequency (SIF)**.

   where $\boldsymbol{w}_i$ is the word vector for the ith word in the sentence, a is a parameter, and $p(\boldsymbol{w}_i)$ is the (estimated) word frequency [over the entire corpus].
2. Remove the projections of the average vectors on their first principal component ("common component removal") (y tho?).

1

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

```
for all sentence s in S do
    v_s ← (1/|s|) Σ_{w∈s} (a/(a+p(w))) v_w
end for
Compute the first principal component u of {v_s : s ∈ S}
for all sentence s in S do
    v_s ← v_s − u u^⊤ v_s
end for
```

Further Reading.

- Learning representations by back-propagating errors (Rumelhard et al., 1986)
- A Neural Probabilistic Language Model (Bengio et al., 2003)
- NLP (almost) from Scratch (Collobert & Watson, 2008)
- Word2Vec (Miklov et al. 2013)

GloVe (Lec 3)

Written by Brandon McKinzie

Skip-gram and negative sampling. Main idea:

- Split the loss function from last lecture into two (additive) terms corresponding to the numerator and denominator respectively (you've done this a trillion times).
- The second term is an expectation over all the words in your vocab space. That is huge, so instead we only use a subsample of size k (the negative samples).
- Interpretation: First term is maximizing $\Pr(o \mid c)$, the probability that the true outside word (given by index o) occurs given context (index) c. Second term is minimizing the probability of random words (the negative samples) occurring around the center (context) word given by c.

To sample the negative samples, draw from $P(w) = U(w)^{3/4}/Z$, where U is the **unigram distribution**.

GloVe (Global Vectors). Given some co-occurrence matrix we computed with previous methods, we can use the following GloVe loss function over all pairs of co-occurring words in our matrix:

$$J(\theta) = \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2 \tag{593}$$

where P_{ij} is computed simply from the counts of words i and j co-occurring (empirical probability) and f is some squashing function that really isn't discussed in this lecture (**TODO**).

Evaluating word vectors.

- **Word Vector Analogies**: Basically, determining if we can do standard analogy fill-in-the-blank problems: "*man [a] is to woman [b] as king [c] is to <blank>*" (if you answered "queen", you'd make a good AI). We can determine this using a standard cosine distance measure:

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||} \tag{594}$$

  Woah that is pretty neat. The solution is x_i = queen. $x_b - x_a$ is the vector pointing from man to woman, which encodes the type of similarity we are looking for with the other pair. Therefore, we take the vector to "king" and *add* the aforementioned difference vector – the resultant vector should point to "queen". Neat!

Derivation. Based on the descriptions in the original paper[139] We want to develop a model for learning word vectors.

1. The authors argue that "the appropriate starting point for word vector learning should be with ratios of co-occurrence probabilities rather than the probabilities themselves." The most general such model takes the form,

$$F(w_i, w_j, \widetilde{w}_k) = \frac{\Pr\left[\widetilde{w}_k \mid w_i\right]}{\Pr\left[\widetilde{w}_k \mid w_j\right]} \equiv \frac{P_{ik}}{P_{jk}} \quad \text{where all} \quad w \in \mathbb{R}^d \tag{595}$$

   and the tilde in $\widetilde{w}_k$ denotes that $\widetilde{w}_k$ is a *context* word vector, which are given a distinct space from the word vectors w_i and w_j being compared. We compute all P_{ik} via frequency counts over the corpus.
2. Now that we've specified the inputs and ratio of interest, we can start specifying some desirable constraints on the function F that we're trying to find. The authors speculate that, since vector spaces are linear structures, we should have F encode the information of the ratio in the vector space via <u>vector differences</u>:

$$F(w_i - w_j, \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{596}$$

   which basically says "our representation of the word vectors should be s.t. the **relative** probability of some word $\widetilde{w}_k$ occurring in the context of a word w_i compared to it occurring in the context of a different word w_j can be captured by $w_i - w_j$ and $\widetilde{w}_k$ alone."
3. Next we notice that F maps arguments in $\mathbb{R}^d$ to a scalar in $\mathbb{R}$. The most straightforward way of doing so while maintaining the linear structure we are trying to capture is via a dot product:

$$F((w_i - w_j)^T \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{597}$$

   Note that now $F : \mathbb{R} \mapsto \mathbb{R}$.
4. We want our model to be invariant under the exchanges $w \leftrightarrow \widetilde{w}$ and $X \leftrightarrow X^T$. We can restore this symmetry by first requiring that F be a **homomorphism**[140] between the groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$ (in our case, negation and division would be better symbols, but it's equivalent). This requires that the following relation hold[141]

$$F(w_i^T \widetilde{w}_k - w_j^T \widetilde{w}_k) = \frac{F(w_i^T \widetilde{w}_k)}{F(w_j^T \widetilde{w}_k)} \tag{598}$$

   where I've grouped terms on the LHS to emphasize how this is the definition of homomorphism. The solution for this equation is that $F(\cdot) \equiv \exp(\cdot)$. Combining this realization with equations 597 and 598 yields,

$$F(w_i^T \widetilde{w}_k) = e^{w_i^T \widetilde{w}_k} = P_{ik} \tag{599}$$

$$\Rightarrow w_i^T \widetilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i) \tag{600}$$

[139] Pennington et al., "GloVe: Global Vectors for Word Representation."

[140] In more detail, $F : (\mathbb{R}, +) \mapsto (\mathbb{R}_{>0}, \times)$, which reads "the function F maps elements in $\mathbb{R}$ and any summation of elements in $\mathbb{R}$ to elements in $\mathbb{R}$ that are greater than zero or any product of positive elements in $\mathbb{R}$."

[141] Note that we do not need to know anything about the RHS of the equations above to state this relation. We write it by definition of homomorphism.

where X_{ik} is the number of times word k appears in the context of word i, and $X_i = \sum_k X_{ik}$ is the number of times any word appears in the context of i.

5. Restore symmetry under the exchanges $w \leftrightarrow \widetilde{w}$ and $X \leftrightarrow X^T$. We absorb X_i into a bias b_i for w_i since it is independent of k.

$$w_i^T \widetilde{w}_k + b_i + \widetilde{b}_k = \log(X_{ik}) \tag{601}$$

6. A main drawback to this model is that it weighs all co-occurrences equally, even those that happen rarely or never. The authors propose a new weighted least squares regression model, introducing a weighting function $f(X_{ij})$ into the cost function of our model:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b}_j - \log X_{ij})^2 \tag{602}$$

Speech and Language Processing

Contents

6.1 Introduction (Ch. 1 2nd Ed.) 208
6.2 Morphology (Ch. 3 2nd Ed.) 209
6.3 N-Grams (Ch. 6 2nd Ed.) 210
6.4 Naive Bayes and Sentiment (Ch. 6 3rd Ed.) 212
6.5 Hidden Markov Models (Ch. 9 3rd Ed.) 214
6.6 POS Tagging (Ch. 10 3rd Ed.) 217
6.7 Formal Grammars (Ch. 11 3rd Ed.) 220
6.8 Vector Semantics (Ch. 15) 221
6.9 Semantics with Dense Vectors (Ch. 16) 224
6.10 Information Extraction (Ch. 21 3rd Ed) 227

Speech and Language Processing **July 10, 2017**

Introduction (Ch. 1 2nd Ed.)

Table of Contents Local *Written by Brandon McKinzie*

Overview. Going to rapidly jot down what seems most important from this chapter.

- **Morphology**: captures information about the shape and behavior of words in context (Ch. 2/3).
- **Syntax**: knowledge needed to order and group words together.
- **Lexical semantics**: knowledge of the meanings of individual words.
- **Compositional semantics**: knowledge of how these components (words) combine to form larger meanings.
- **Pragmatics**: the appropriate use of polite and indirect language.
- The knowledge of language needed to engage in complex language behavior can be separated into the following 6 distinct categories:
  1. Phonetics and Phonology – The study of linguistic sounds.
  2. Morphology – The study of the meaningful components of words.
  3. Syntax – The study of the structural relationships between words.
  4. Semantics – The study of meaning.
  5. Pragmatics – The study of how language is used to accomplish goals.
  6. Discourse – The study of linguistic units larger than a single utterance.
- Methods for **resolving ambiguity**: pos-tagging, word sense disambiguation, probabilistic parsing, and speech act interpretation.
- **Models and Algorithms**. Among the most important are **state space search** and **dynamic programming** algorithms.

Morphology (Ch. 3 2nd Ed.)

 Written by Brandon McKinzie

English Morphology. Morphology is the study of the way words are built up from smaller meaning- bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language[142]. The two classes of morphemes are **stems** (the "main" morpheme of the word) and **affixes** (the "additional" meanings of various kinds).

Affixes are further divided into prefixes (precede stem), suffixes (follow stem), circumfixes (do both), and infixes (inside the stem).

Two classes of ways to form words from morphemes: **inflection** and **derivation**. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. Derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a different class, often with a meaning hard to predict exactly.

[142] Examples: "fox" is its own morpheme, while "cats" consists of the morpheme "cat" and the morpheme "-s".

N-Grams (Ch. 6 2nd Ed.)

 Written by Brandon McKinzie

Counting Words. Most N-gram based systems deal with the *wordform*, meaning they treat words like "cats" and "cat" as distinct. However, we may want to treat the two as instances of a single abstract word, or **lemma**: a set of lexical forms having the same stem, the same major part of speech, and the same word-sense.

Simple (Unsmoothed) N-Grams. An N-gram is a N-1th order Markov model (because it looks N-1 steps in the past). Notation: the authors use the convention that $w_1^n \triangleq w_1, w_2, \ldots, w_n$ to denote a sequence of n words. Given this, we can write the general equation for the N-gram approximation for the probability of the nth word ($n > N$) in a sentence:

$$\Pr\left[w_n \mid w_1^{n-1}\right] \approx \Pr\left[w_n \mid w_{n-N+1}^{n-1}\right] \tag{603}$$

for $N \geq 2$. We can compute these probabilities by simply counting:

$$\Pr\left[w_n \mid w_1^{n-1}\right] = \frac{C(w_{n-N+1}^{n-1} w_n)}{C(w_{n-N+1}^{n-1})} \tag{604}$$

where $C(\cdot)$ is the number of times the sequence, denoted as $\cdot$, occurred in the corpus.

Entropy. Denote the random variable of interest as $\mathbf{x}$ with probability function $p(\mathbf{x})$. The entropy of this random variable is:

$$H(\mathbf{x}) = -\sum_x p(\mathbf{x} = x) \log_2 p(\mathbf{x} = x) \tag{605}$$

which should be thought of as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme. The value 2^H is the **perplexity**, which can be interpreted as the weighted average number of choices a random variable has to make.

Cross Entropy for Comparing Models. Useful when we don't know the actual probability distribution p that generated some data. Assume we have some model m that's an approximation of p. The cross-entropy of m on p is defined by:

$$H(p, m) = \lim_{n \to \infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log m(w_1, \ldots, w_n) \tag{606}$$

That is we draw sequences according to the probability distribution p, but sum the log of their probability according to m. **TODO**: continue off at pg 250/975 of textbook.

Naive Bayes and Sentiment (Ch. 6 3rd Ed.)

 Written by Brandon McKinzie

[3rd Ed.] [Quick Review]

Overview. This chapter is concerned with **text categorization**, the task of classifying an entire text by assigning it a label drawn from some set of labels. *Generative classifiers* like naive Bayes build a model of each class. Given an observation, they return the class most likely to have generated the observation. *Discriminative classifiers* like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. Notation: we will assume we have a training set of N documents, each hand-labeled with some class: $\{(d_1, c_1), \ldots, (d_N, c_N)\}$.

Discriminative systems are often more accurate and hence more commonly used.

Naive Bayes. A multinomial[143] classifier with a naive assumption about how the features interact. We model a text document as a **bag of words**, meaning we store (1) the words that occurred and (2) their frequencies. It's a probabilistic classifier, meaning it estimates the label/class of a document d as

$$\hat{c} = \arg\max_{c \in C} \Pr[c \mid d] \tag{608}$$

$$= \arg\max_{c \in C} \frac{\Pr[d \mid c] \Pr[c]}{\Pr[d]} \tag{609}$$

$$= \arg\max_{c \in C} \Pr[d \mid c] \Pr[c] \tag{610}$$

Computing the class-conditional distribution (the likelihood) $\Pr[d \mid c]$ over all possible $d \in D$ is typically intractable; we must introduce some simplifying assumptions and use an approximation of it. Our assumptions in this section will be:

- **Bag-of-Words**: Assume that word position within a document is irrelevant. (Counts still matter)

143

Rapid review: **multinomial distribution**. Let $\mathbf{x} = (x_1, \ldots, x_k)$ denote the result of an experiment with n independent trials ($n = \sum_i^k x_i$) and k possible outcomes for any given trial. i.e. x_i is the number of trials that had outcome i ($1 \leq i \leq k$). The pmf of this multinomial distribution, over all possible $\mathbf{x}$ constrained by $n = \sum_i^k x_i$, is:

$$\Pr[\mathbf{x} = (x_1, \ldots, x_k); n] = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \times \cdots \times p_k^{x_k} \tag{607}$$

where p_i is the probability of outcome i for any single trial.

- **Naive Bayes Assumption**: First, recall that d is typically modeled as a (random) vector consisting of features $f_1, \ldots, f_n$, each of which has an associated probability distribution $\Pr[f_i \mid c]$. The NB assumption is that the features are mutually independent given the class c:

$$\Pr[f_1, \ldots, f_n \mid c] = \Pr[f_1 \mid c] \cdots \Pr[f_n \mid c] \tag{611}$$

$$c_{NB} = \arg\max_{c \in C} \Pr[c] \prod_{f \in F} \Pr[f \mid c] \tag{612}$$

  where 612 is the final equation for the class chosen by the naive Bayes classifier.

In text classification we typically use the word at position i in the document as f_i, and move to log space to avoid underflow/increase speed:

$$c_{NB} = \arg\max_{c \in C} \log \Pr[c] + \sum_{i}^{len(d)} \log \Pr[w_i \mid c] \tag{613}$$

Classifiers that use a linear combination of the inputs to make a classification decision (e.g. NB, logistic regression) are called linear classifiers.

Training the NB Classifier. No real "training" in my opinion, just simple counting from the data:

$$\hat{P}[c] = \frac{N_c}{N_{docs}} \tag{614}$$

$$\hat{P}[w_i \mid c] = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|} \tag{615}$$

The Laplace smoothing is added to avoid the occurrence of zero-probabilities in equation 612.

Optimizing for Sentiment Analysis.

- It often improves performance [for sentiment] to clip word counts in each document to 1 (“binary NB”).
- deal with *negations* in some way.
- Use sentiment lexicons, lists of words that are pre-annotated with positive or negative sentiment.

Hidden Markov Models (Ch. 9 3rd Ed.)

 Written by Brandon McKinzie

Overview. Here we will first go over the math behind HMMs: the **Viterbi**, **Forward**, and **Baum-Welch** (EM) algorithms for unsupervised or semi-supervised learning. Recall that a HMM is defined by specifying the set of N states Q, transition matrix A, sequence of T observations O, sequence of observation likelihoods B, and the initial/final states. They can be characterized by three fundamental problems:

1. **Likelihood**. Given an HMM $\lambda = (A, B)$ and observation sequence, compute the likelihood (prob. of the observations given the model). (**Forward**)
2. **Decoding**. Given an HMM $\lambda = (A, B)$ and observation sequence, discover the best hidden state sequence. (**Viterbi**)
3. **Learning**. Given an observation sequence and the set of states in the HMM, learn the HMM parameters A and B. (**Baum-Welch/Forward-Backward/EM**)

The Forward Algorithm. For likelihood computation. We want to compute the probability of some sequence of observations O, without knowing the sequence of hidden states (that emitted the observations) Q. In general, this can be expressed by summing over all possible hidden state sequences:

$$\Pr[O] = \sum_{Q} \Pr[Q] \Pr[O \mid Q] \tag{616}$$

However, for N hidden states and T observations, this summation involves N^T terms, which becomes intractable rather quickly. Instead, we can use the $\mathcal{O}(N^2T)$ **Forward Algorithm**. The forward algorithm can be defined by initialization, recursion definition, and termination, shown respectively as follows:

$$\alpha_1(j) = a_{0j} b_j(o_1) \qquad 1 \le j \le N \tag{617}$$

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \quad 1 \le j \le N, 1 \le t \le T \tag{618}$$

$$\Pr[O \mid \lambda] = \alpha_T(q_F) = \sum_{i=1}^{N} \alpha_T(i) a_{iF} \tag{619}$$

Viterbi Algorithm. For decoding. Want the most likely hidden state sequence given observations. Let $v_t(j)$ represent the probability that we are in state j after t observations and passing through the most probable state sequence $q_0, q_1, \ldots, q_{t-1}$. Similar to the forward algorithm, we show the defining equations for the Viterbi algorithm below:

$$v_1(j) = a_{0j} b_j(o_1) \quad 1 \leq j \leq N \tag{620}$$

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(j) a_{ij} b_j(o_t) \tag{621}$$

$$P^* = v_T(q_F) = \max_{i=1}^{N} v_T(i) a_{iF} \tag{622}$$

N.B.: At each step, the best path up to that point can be found by taking the *argmax* instead of max.

Baum-Welch Algorithm. AKA forward-backward algorithm, a special case of the EM algorithm. Given an observation sequence O and the set of best possible states in the HMM, learn the HMM parameters A and B. First, we must define some new notation. The **backward probability** β is defined as:

$$\beta_t(i) \triangleq \Pr\left[o_{t+1}, \ldots, o_T \mid q_t = i, \lambda\right] \tag{623}$$

Remember $\lambda \equiv (A, B)$

As usual, we can compute its values inductively as follows:

$$\beta_T(i) = a_{iF} \qquad 1 \leq i \leq N \tag{624}$$

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \quad 1 \leq i \leq N, 1 \leq t \leq T \tag{625}$$

$$\Pr\left[O \mid \lambda\right] = \alpha_T(q_F) = \beta_1(q_0) \tag{626}$$

$$= \sum_{j=1}^{N} a_{0j} b_j(o_1) \beta_1(j) \tag{627}$$

We can use the forward and backward probabilities α and β to compute the transition probabilities a_{ij} and observation probabilities $b_i(o_t)$ from an observation sequence. The derivation steps are as follows:

1. **Estimating** $\hat{a}_{ij}$. Begin by defining quantities that will prove useful:

Remember, knowing the observation sequence does NOT give us the sequence of <u>hidden</u> states.

$$\xi_t(i,j) \triangleq \Pr\left[q_t = i, q_{t+1} = j \mid O, \lambda\right] \tag{628}$$

$$\widetilde{\xi}_t(i,j) \triangleq \Pr\left[q_t = i, q_{t+1} = j, O \mid \lambda\right] \tag{629}$$

$$= \alpha_t(i) a_{ij} b_j(t+1) \beta_{t+1}(j) \tag{630}$$

where you should be able to derive eq. 630 in your head using just logic. If you cannot, review before continuing. We can then derive $\xi_t(i,j)$ using basic definitions of conditional probability, combined with eq. 626. Finally, we estimate $\hat{a}_{ij}$ as the expected number of transitions $q_i \to q_j$ divided by the expected number of transitions from q_i total:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)} \tag{631}$$

2. **Estimating** $\hat{b}_j(v_k)$. We define our estimate as the expected number of times we are in q_j and emit observation v_k, divided by the expected number of times we are in state j. Similar to our approach for $\hat{a}_{ij}$ we define helper quantities for these values at a given timestep, then sum over them (all t) to obtain our estimate.

$$\gamma_t(j) \triangleq \Pr\left[q_t = j \mid O, \lambda\right] \tag{632}$$

$$= \frac{\Pr\left[q_t = j, O \mid \lambda\right]}{\Pr\left[O \mid \lambda\right]} \tag{633}$$

$$= \frac{\alpha_t(j)\beta_t(j)}{\Pr\left[O \mid \lambda\right]} \tag{634}$$

Thus, we obtain $\hat{b}_j(v_k)$ by summing over all timesteps where $o_t = v_k$, denoted as the set T_{v_k}, divided by the summation over all t regadless of o_t:

$$\hat{b}_j(v_k) = \frac{\sum_{t \in T_{v_k}} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{635}$$

At last we can finally define the **Forward-Backward Algorithm** as follows:

1. Initialize A and B.
2. **E-step**. Compute $\gamma_t(j)$ $(\forall t, j)$, and compute $\xi_t(i,j)$ $(\forall t, i, j)$.
3. **M-step**. Update all $\hat{a}_{ij}$ and $\hat{b}_j(v_k)$.
4. Upon convergence, return A and B.

POS Tagging (Ch. 10 3rd Ed.)

 Written by Brandon McKinzie

English Parts-of-Speech. POS are traditionally defined based on syntactic and morphological function, grouping words that have similar neighboring words or take similar affixes.

	Part of Speech	Definition	Properties
Open classes:	noun	people, places, things	occur with determiners, take possessives,
	verb	actions, processes	3rd-person-sg, progressive, past participle
	adjective	properties, qualities	
	adverb	modify verbs, adverbs, verb phrases	directional, locative, degree, manner, tem

Common nouns can be divided into *count* (e.g. goat/goats) and *mass* (e.g. snow) nouns.

Closed classes. POS with relatively fixed membership. Some of the most important in English are:

prepositions: on, under, over, near, by, at, from, to, with
determiners: a, an, the
pronouns: she, who, I, others
conjunctions: and, but, or, as, if, when
auxiliary verbs: can, may, should, are
particles: up, down, on, off, in, out, at, by
numerals: one, two, three, first, second, third

Some subtleties: the **particle** resembles a preposition or an adverb and is used in combination with a verb. An example case where "over" is a particle: "she turned the paper over." When a verb and a particle behave as a single syntactic and/or semantic unit, we call the combination a **phrasal verb**. Phrasal verbs cause widespread problems with NLP because they often behave as a semantic unit with a noncompositional meaning – one that is not predictable from the distinct meanings of the verb and the particle. Thus, "turn down" means something like "reject", "rule out" means "eliminate", "find out" is "discover", and "go on" is "continue".

HMM POS Tagging. Since we typically train on labeled data, we need only use the Viterbi algorithm for decoding[144]. In the POS case, we wish to find the sequence of n tags, $\hat{t}_1^n$, given the observation sequence of n words w_1^n.

$$\hat{t}_1^n = \arg\max_{t_1^n} \Pr\left[t_1^n \mid w_1^n\right] = \arg\max_{t_1^n} \Pr\left[w_1^n \mid t_1^n\right] \Pr\left[t_1^n\right] \tag{636}$$

where we've dropped the denominator after using Bayes' rule (since argmax is the same). HMM taggers made two further simplifying assumptions:

$$\Pr\left[w_1^n \mid t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \tag{637}$$

$$\Pr\left[t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[t_i \mid t_{i-1}\right] \tag{638}$$

We can thus plug-in these values into eq. 636 to obtain the equation for $\hat{t}_1^n$.

$$\hat{t}_1^n = \arg\max_{t_1^n} \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \Pr\left[t_i \mid t_{i-1}\right] \tag{639}$$

$$= \arg\max_{t_1^n} \prod_{i=1}^{n} b_i(w_i) a_{i-1,i} \tag{640}$$

where I've written a "translated" version on the second line using the familiar syntax from the previous chapter. In practice, we can obtain quick estimates for the two probabilities on the RHS by taking counts/averages over our tagged training data. We then run through the Viterbi algorithm to find all the argmaxes over states for the most likely hidden state sequence.

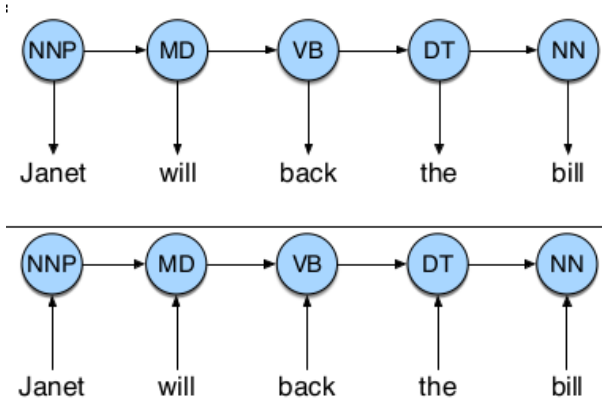
Maximum Entropy Markov Models (MEMMs). A sequence model adaptation of the MaxEnt (multinomial logistic regression) classifier[145]. Since HMMs are generative models, they decompose $\Pr[T \mid W]$ into $\Pr[W \mid T] \Pr[T]$ when computing the best tag sequence $\hat{T}$. Since MEMMs are discriminative, they compute/model $\Pr[T \mid W]$ directly:

$$\hat{T} = \arg\max_{T} \Pr\left[T \mid W\right] = \arg\max_{T} \prod_{i} \Pr\left[t_i \mid w_i, t_{i-1}\right] \tag{641}$$

Visually, we can think of the difference between HMMs and MEMMs via the direction of arrows, as illustrated below.

[144] Recall that decoding is the problem of finding the best hidden state sequence, given $\lambda = (A, B)$ and observation sequence O.

[145] Because it is based on logistic regression, the MEMM is a **discriminative sequence model**. By contrast, the HMM is a **generative sequence model**.



The top shows the HMM representation, while the bottom is MEMM.

> *The reason to use a discriminative sequence model is that discriminative models make it easier to incorporate a much wider variety of features.*

Bidirectionality. The one problem with the MEMM and HMM models as presented is that they are exclusively run left-to-right. MEMMs[146] have a weakness known as the **label bias problem**. Consider the tagged fragment: "*will/NN to/TO fight/VB*[147]." Even though the word "*will*" is followed by "*to*", which strongly suggests "*will*" is a NN, a MEMM will incorrectly label "*will*" as MD (modal verb). The culprit lies in the fact that $\Pr[\text{TO} \mid to, t_{will}]$ is essentially 1 regardless of t_{will}; i.e. the fact that "*to*" must have the tag TO has **explained away** the presence of TO and so the model doesn't learn the importance of the previous NN tag for predicting TO.

One way to implement bidirectionality (and thus allowing e.g. the link between TO being available when tagging the NN) is to use a **Conditional Random Field** (CRF) model. However, CRFs are much more computationally expensive than MEMMs and don't work better for tagging.

[146] And other non-generative finite-state models based on next-state classifiers

[147] Note on the tag meanings: TO literally means "to". MD means "modal" and refers to modal verbs such as *will*, *shall*, etc.

Constituency and CFGs. Discovering the inventory of constituents present in the language. Groups of words like *noun phrases* or *prepositional phrases* can be thought of as single units which can only be placed within certain parts of a sentence.

The most widely used formal system for modeling constituent structure in English is the **Context-Free Grammar**[148]. A CFG consists of a set of **productions** (rules), e.g.

$$\text{NP} \longrightarrow \text{Det Nominal} \tag{642}$$

$$\text{NP} \longrightarrow \text{ProperNoun} \tag{643}$$

$$\text{Nominal} \longrightarrow \text{Noun} \mid \text{Nominal Noun} \tag{644}$$

where the arrow is to be read "is composed of" or "consists of."

The sequence of rule expansions going from left to right is called a **derivation** of the string of words, commonly represented by a **parse tree**. The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**.

[148] Also called Phrase-Structure Grammars. Equiv formalism as Backus-Naur Form (BNF)

Words and Vectors. Vector models are generally based on a **co-occurrence**, an example of which is a **term-document matrix**: Each row is identified by a word, and each column a document. A given cell value is the number of times the assoc. word occurred in the assoc. document. Can also view each column as a document vector.

Information Retrieval: task of finding document d, from D docs total, that best matches a query q.

For individual word vectors, however, it is most common to instead use a **term-term** matrix[149], in which columns are also identified by individual words. Now, cell values are the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus. The context is most commonly a window around the row/target word, meaning a cell gives the number of times the column word occurs in a window of $\pm N$ words from the row word.

- **Q**: What about the co-occurrence of a word with itself (row i, col i)?
  - **A**: It is included, yes. Source: "The size of the window ... is generally between 1 and 8 words on each side of the target word (*for a total context of 3-17 words*)."
- **Q**: Why is the size of each vector generally $|V|$ (vocab size)? Shouldn't this vary substantially with window and corpus size?
  - **A**: idk

(**TODO**: revisit end of page 5 in my pdf of this)

Pointwise Mutual Information (PMI). Motivation: raw frequencies in a co-occurrence matrix aren't that great, and words like "the" (which aren't useful and occur everywhere) can really skew things. *The best weighting or measure of association between words should tell us how much more often than chance the two words co-occur.* PMI is such a measure.

$$\textbf{[Mutual Information]} \qquad I(X,Y) = \sum_x \sum_y P(X=x, Y=y)\text{PMI}(x,y) \tag{645}$$

$$\textbf{[PMI]} \qquad \text{PMI}(x,y) = \ln \frac{P(x,y)}{P(x)P(y)} \tag{646}$$

which can be applied for our specific use case as $\text{PMI}(w,c) = \ln \frac{P(w,c)}{P(w)P(c)}$. The interpretation is simple: the denominator tells the joint probability of the given target word w occurring with context word c if they were independent of each other, while the numerator tells us how often we observed the two words together (assuming we compute probability by using the MLE).

[149] Also called the word-word or term-context matrix

Therefore, the ratio gives us how an estimate of how much more the target and feature co-occur than we expect by chance[150]. Most people use **Positive PMI**, which is just $\max(0, \text{PMI})$. We can compute a **PPMI matrix** (to replace our co-occurrence matrix), where PPMI_{ij} gives the PPMI value of word w_i with context c_j. The authors show a few formulas which is really distracting since all we actually need is the counts $f_{ij} = \text{counts}(w_i, c_j)$, and from there we can use basic probability and Bayes rule to get the PPMI formula.

- **Q**: Explain why the following is true: very rare words tend to have very high PMI values.
  - **A**: hi
- **Q**: What is the range of α used in PPMI_α? What is the intuition behind doing this?
  - **A**: For reference:

$$\text{PPMI}_\alpha(w, c) = \max\left(\ln \frac{P(w, c)}{P(w)P_\alpha(c)}, 0\right) \tag{647}$$

$$P_\alpha(c) = \frac{\text{count}(c)^\alpha}{\sum_{c'} \text{count}(c')^\alpha} \tag{648}$$

Although there are better methods than PPMI for weighted co-occurrence matrices, most notably **TF-IDF**, things like tf-idf are <u>not</u> generally used for measuring *word similarity*. For that, PPMI and significance-testing metrics like t-test and likelihood-ratio are more common. The **t-test** statistic, like PMI, measures how much more frequent the association is than chance.

$$t = \frac{\bar{x} - \mu}{\sqrt{s^2/N}} \tag{649}$$

$$\text{t-test}(a, b) = \frac{P(a, b) - P(a)P(b)}{\sqrt{P(a)P(b)}} \tag{650}$$

where $\bar{x}$ is the observed mean, while μ is the expected mean [under our null-hypothesis of independence].

Measuring Similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors:

$$\text{cosine}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\boldsymbol{v} \cdot \boldsymbol{w}}{|\boldsymbol{v}|\ |\boldsymbol{w}|} \tag{651}$$

Note that, since we've been defining vector elements as frequencies/PPMI values, they won't have negative elements, and thus our cosine similarities will be between 0 and 1 (not -1 and 1).

[150] Computing PMI this way can be problematic for word pairs with small probability, especially if we have a small corpus. Recognize that PMI should never really be negative, but in practice this happens for such cases

Alternatives to cosine:

- **Jaccard measure**: Described as "weighted number of overlapping features, normalized", but looks like a silly hack in my opinion:

$$\text{sim}_{Jac}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} \max(\boldsymbol{v}_i, \boldsymbol{w}_i)} \tag{652}$$

- **Dice measure**: Another hack. This displeases me.

$$\frac{2 \times \sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} (\boldsymbol{v}_i + \boldsymbol{w}_i)} \tag{653}$$

- **Jensen-Shannon Divergence**: An alternative to the KL-divergence[151], which represents the divergence of each distribution from the mean of the two:

$$\text{sim}_{JS}(\boldsymbol{v}||\boldsymbol{w}) = D\left(\boldsymbol{v}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) + D\left(\boldsymbol{w}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) \tag{655}$$

[151] Idea:if two vectors, $\boldsymbol{v}$ and $\boldsymbol{w}$, each express a probability distribution (their values sum to one), then they are are similar to the extent that these probability distributions are similar. The basis of comparing two probability distributions P and Q is the **Kullback-Leibler** divergence or relative entropy, defined as:

$$D(P||Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)} \tag{654}$$

Semantics with Dense Vectors (Ch. 16)

Overview. This chapter introduces three methods for generating short, dense vectors: (1) dimensionality reduction like SVD, (2) neural networks like skip-gram or CBOW, and (3) Brown clustering.

Dense Vectors via SVD. Method for finding more important dimensions of a dataset, "important" defined as dimensions wherein the data most varies. First applied (for language) for generating embeddings from term-document matrices in a model called **Latent Semantic Analysis** (LSA). LSA is just SVD on a $|V| \times c$ term-document matrix $\boldsymbol{X}$, factorized into $\boldsymbol{W\Sigma C}^T$. By using only the top $k < m$ dimensions of these three matrices, the product becomes a least-squares approx. to the original X. It also gives us the reduced $|V| \times k$ matrix $\boldsymbol{W}_k$, where each row (word) is a k-dimensional vector (embedding). Voilà, we have our dense vectors!

$\boldsymbol{W} \in \mathbb{R}^{|V| \times m}$
$\boldsymbol{\Sigma} \in \mathbb{R}^{m \times m}$
$\boldsymbol{C}^T \in \mathbb{R}^{m \times c}$

Note that LSA implementations typically use a particular weighting of each cell in the term-document matrix called the **local** and **global** weights.

$$\text{[local]} \qquad \log \mathrm{f}(i,j) + 1 \tag{656}$$

$$\text{[global]} \qquad 1 + \frac{\sum_j p(i,j) \log p(i,j)}{\log D} \tag{657}$$

$f(i,j)$ is the raw frequency of word i in context j. D is number of docs.

For the case of a word-word matrix, it is common to use PPMI weighting.

Skip-Gram and CBOW. Neural models learn an embedding by starting with a random vector and then iteratively shifting a word's embeddings to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby[152] Word2vec, for example, learns embeddings by training to predict neighboring words[153].

- **Skip-Gram**: Learns two embeddings for each word w: the **word embedding** v (within matrix $\boldsymbol{W}$) and **context embedding** c (within matrix $\boldsymbol{C}$). Visually:

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{v}_0^T \\ \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_{|V|}^T \end{pmatrix} \qquad \boldsymbol{C} = \begin{pmatrix} \boldsymbol{c}_0 & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_{|V|} \end{pmatrix} \tag{658}$$

[152] Why? Why is this a sensible assumption? I see no reason a priori why it ought to be true.
[153] Note that the prediction task is not the goal – it just happens to result in good word embeddings. Hacky.

For a context window of $L = 2$, and at a given word $\boldsymbol{v}^{(t)}$ inside the corpus[154], our goal is to predict the context [words] denoted as $\left[\, \boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)} \,\right]$.

- Example: Consider one of the context words, say $\boldsymbol{c}^{(t+1)} \triangleq \boldsymbol{c}_k$, where we also assume it's the kth word in our vocab. Also assume that our target word $\boldsymbol{v}^{(t)} \triangleq \boldsymbol{v}_j$ is the jth word in our vocab.
- Our task is to compute $\Pr\left[\boldsymbol{c}_k \mid \boldsymbol{v}_j\right]$. We do this with a softmax:

$$\Pr\left[\boldsymbol{c}_k \mid \boldsymbol{v}_j\right] = \frac{e^{\boldsymbol{c}_k^T \boldsymbol{v}_j}}{\sum_{i \in |V|} e^{\boldsymbol{c}_i^T \boldsymbol{v}_j}} \tag{659}$$

- **CBOW**: Continuous bag of words. Basically the mirror-image of skip-gram. Goal is to predict current word $\boldsymbol{v}^{(t)}$ from the context window of 2L words $\left[\, \boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)} \,\right]$.

As usual, the denominator of the softmax is computationally expensive, and usually we approximate it with **negative sampling**.

> *In the training phase, the algorithm walks through the corpus, at each target word choosing the surrounding context words as positive examples, and for each positive example also choosing k noise samples or negative samples: non-neighbor words. The goal will be to move the embeddings toward the neighbor words and away from the noise words.*

Example: Suppose we come along the following window (in "[]") (L=2) in our corpus:

```
lemon, a [tablespoon of apricot preserves or] jam
```

Ultimately, we want dot products, $\boldsymbol{c}_i \cdot vector(\text{"apricot"})$, to be <u>high</u> for all four of the context words $\boldsymbol{c}_i$. We do negative sampling by sampling k random noise words according to their [unigram] frequency. So here, for e.g. $k = 2$, this would amount to 8 noise words, 2 for each context word. We want the dot products between "apricot" and these noise words to be <u>low</u>. For a given single context-word pair (w, c), our training objective is to maximize:

$$\log \sigma(c \cdot w) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim p(w)} \left[\log \sigma(-w_i \cdot w)\right] \tag{660}$$

In practice, common to use $p^{3/4}(w)$ instead of $p(w)$

Again, the above is for a single context-target word-pair and, accordingly, the summation is only over $k = 2$ (for our example). Don't try to split the expectation into a summation or anything – just view it as an expected value. To iteratively shift parameters, we use an optimizer like SGD.

[154] Note that, technically, the position t of $\boldsymbol{v}^{(t)}$ is irrelevant for our computation; we are predicting those words based on which word $\boldsymbol{v}^{(t)}$ is in the vocabulary, not it's position in the corpus.

The actual model architecture is a typical neural net, progressing as follows:

$$
\begin{aligned}
\text{“apricot”} \;&\rightarrow\; \boldsymbol{w}^{\text{one-hot}} = \left[\, 0 \; 0 \; \cdots \; 1 \; \cdots \; 0 \,\right] && (661)\\
&\rightarrow\; \boldsymbol{h} = \boldsymbol{W}^T \boldsymbol{w}^{\text{one-hot}} && (662)\\
&\rightarrow\; \boldsymbol{o} = \boldsymbol{C}^T \boldsymbol{h} = \left[\, \boldsymbol{c}_0^T \boldsymbol{h},\, \boldsymbol{c}_1^T \boldsymbol{h},\, \cdots\; \boldsymbol{c}_{|V|}^T \boldsymbol{h} \,\right]^T && (663)\\
&\rightarrow\; \boldsymbol{y} = \text{softmax}(\boldsymbol{o}) = \left[\, \Pr[\boldsymbol{c}_0|\boldsymbol{h}],\, \Pr[\boldsymbol{c}_2|\boldsymbol{h}],\, \cdots\; \Pr\left[\boldsymbol{c}_{|V|}|\boldsymbol{h}\right] \,\right]^T && (664)\\
& && (665)
\end{aligned}
$$

Brown Clustering. An agglomerative clustering algorithm for deriving vector representations of words by clustering words based on their associations with the preceding or following words. Makes use of the **class-based language model** (CBLM), wherein each word w belongs to some class $c \in C$ via the probability $P(w \mid c)$. CBLMs define

$$P(w_i \mid w_{i-1}) = P(c_i \mid c_{i-1}) P(w_i \mid c_i) \tag{666}$$

$$P(\text{corpus} \mid C) = \prod_{i-1}^{n} P(w_i \mid w_{i-1}) \tag{667}$$

A naive and extremely inefficient version of Brown clustering, a hierarchical clustering algorithm, is as follows:

1. Each word is initially assigned to its own cluster.
2. For each cluster pair $(c_i, c_{j \neq i})$, compute the value of eq 667 that would result from merging c_i and c_j into a single cluster. The pair whose merger results in the <u>smallest decrease</u> in eq 667 is merged.
3. Clustering proceeds until all words are in one big cluster.

This process builds a binary tree from the bottom-up, and the binary string corresp. to a word's traversal from leaf-to-root is its representation.

Information Extraction (Ch. 21 3rd Ed)

 Written by Brandon McKinzie

Overview. The first step in most IE tasks is **named entity recognition** (NER). Next we can do **relation extraction**: finding and classifying semantic relations among the entities, e.g. "spouse-of." **Event extraction** is finding the events in which the entities participate, and **event coreference** for figuring out which event mentions actually refer to the same event. It's also common to extract dates/times (temporal expression) and perform **temporal expression normalization** to map them onto specific calendar dates. Finally, we can do **template filling**: finding recurring/stereotypical situations in documents and filling the template slots with appropriate material.

Named Entity Recognition. Standard algorithm is word-by-word sequence labeling task by a MEMM or CRF, trained to label tokens with tags indicating the presence of particular kinds of NEs. It is common to label with **BIO tagging**, for beginning, inside, and outside of entities. If we have n unique entity types, then we'd have $2n + 1$ BIO tags[155]. A helpful illustration is shown below:

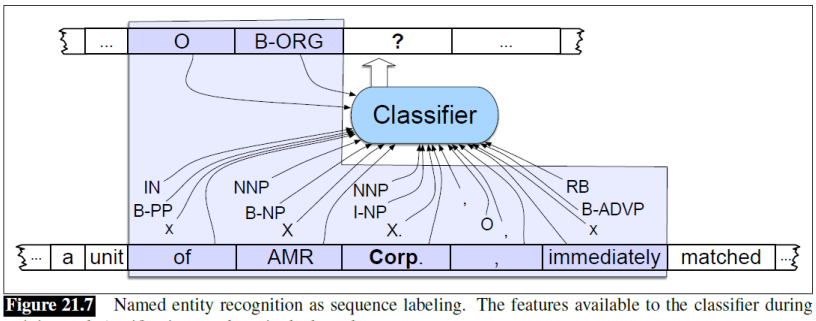


Figure 21.7 Named entity recognition as sequence labeling. The features available to the classifier during training and classification are those in the boxed area.

Here we see a classifier determining the label for *Corp.* with a context window of size 2 and various features shown in the boxed region. For evaluation of NER, we typically use the familiar **recall**, **precision**, and **F1 measure**.

[155] $2n$ for B-<NE> and I-<NE>, with $+1$ for the blanket O tag (not any of our NEs)

Relation Extraction. The four main algorithm classes used are (1) hand-written patterns, (2) supervised ML, (3) semi-supervised, and (4) unsupervised. Terminology:

- **Infobox**: structured tables associated with certain articles/topics/etc. For example, the Wikipedia infobox for Stanford includes structured facts like state = 'California'.
- **Resource Description Framework (RDF)**: a metalanguage of RDF triples, tuples of (entity, relation, entity), called a subject-predicate-object expression. For example: (Golden Gate Park, location, San Francisco).
- **hypernym**: the "is-a" relation.
- **hyponym**: the "kind-of" relation. *Gelidium is a kind of red algae.*

Overview of the four algorithm classes:

1. **Patterns**. Consider a sentence that has the following form:

   NP_0 *such as* $NP_1\{, NP_2, \ldots, (and|or) NP_i\}, i \geq 1$

   also known as a *lexico-syntactic pattern*, which implies $\forall NP_i, i \geq 1$,hyponym(NP_i, NP_0)[156]. Patterns typically have high precision but low-recall.
2. **Supervised**. The general approach for finding relations in a given sequence of words is the following:
   (a) Find all pairs of named entities in the sequence (typically a single sentence).
   (b) For each pair, use a trained binary classifier to predict whether or not the entities in the pair are indeed related.
   (c) If related, use a classifier trained to predict the relation given the entity-pair.

   As with NER, **the most important step** in this process is to identify useful surface features that will be useful for relation classification, including word features, NER features, syntactic paths (chunk seqs, constituent paths, dependency-tree paths), and more.
3. **Semi-supervised** via bootstrapping. Suppose we have a few high-precision **seed patterns** (or seed tuples)[157]. **Bootstrapping** proceeds by taking the entities in the seed pair, and then finding sentences (on the web, or whatever dataset we are using) that contain both entities. From all such sentences, we extract and generalize the context around the entities to learn new patterns.

```
function BOOTSTRAP(Relation R) returns new relation tuples

  tuples ← Gather a set of seed tuples that have relation R
  iterate
    sentences ← find sentences that contain entities in seeds
    patterns ← generalize the context between and around entities in sentences
    newpairs ← use patterns to grep for more tuples
    newpairs ← newpairs with high confidence
    tuples ← tuples + newpairs
  return tuples
```

Figure 21.14 Bootstrapping from seed entity pairs to learn relations.

[156] Here, hyponym(A, B) means "A is a kind-of (hyponym) of B."

[157] seed tuples are tuples of the general form (M1, M2) where M1 and M2 are each specific named entities we know have the relation of interest R.

4. **Unsupervised**. The Re Verb system extracts a relation from a sentence s in 4 steps:

    1. Run a part-of-speech tagger and entity chunker over s
    2. For each verb in s, find the longest sequence of words w that start with a verb and satisfy syntactic and lexical constraints, merging adjacent matches.
    3. For each phrase w, find the nearest noun phrase x to the left which is not a relative pronoun, wh-word or existential "there". Find the nearest noun phrase y to the right.
    4. Assign confidence c to the relation $r = (x, w, y)$ using a confidence classifier and return it.

Event Extraction. An event mention is any expression denoting an event or state that can be assigned to a particular point, or interval, in time. Note that this is quite different than the colloquial usage of the word "event," you should think of the two as distinct. Here, most event mentions correspond to verbs, and most verbs introduce events. Event extraction is typically modeled via ML, detecting events via sequence models with BIO tagging, and assigning event classes/attributes with multi-class classifiers.

Template Filling. The task is creation of one template for each event in the input documents, with the slots filled with text from the document. For example, an event could be "Fare-Raise Attempt" with corresponding template (slots to be filled) "(<Lead Airline>, <Amount>, <Effective Date>, <Follower>)". This is generally modeled by training two separate supervised systems:

1. **Template recognition**. Trained to determine if template T is present in sentence S. Here, "present" means there is a sequence within the sentence that could be used to fill a slot within template T.
2. **Role-filler extraction**. Trained to detect each role (slot-name), e.g. "Lead Airline".

Probabilistic Graphical Models

Contents

7.1 Foundations (Ch. 2) 231

7.1.1 Appendix 233

7.1.2 L-BFGS 236

7.1.3 Exercises 239

7.2 The Bayesian Network Representation (Ch. 3) 241

7.3 Undirected Graphical Models (Ch. 4) 245

7.3.1 Exercises 252

7.4 Local Probabilistic Models (Ch. 5) 254

7.5 Template-Based Representations (Ch. 6) 255

7.6 Gaussian Network Models (Ch. 7) 258

7.7 Variable Elimination (Ch. 9) 259

7.8 Clique Trees (Ch. 10) 263

7.9 Inference as Optimization (Ch. 11) 267

7.10 Parameter Estimation (Ch. 17) 268

7.11 Partially Observed Data (Ch. 19) 270

Foundations (Ch. 2)

(Brief summary of the book's notation, for ease of reference.)

Graphs. Authors denote directed graphs as $\mathcal{G}$ and undirected graphs as $\mathcal{H}$.

- **Induced Subgraph**. Let $\mathcal{K} = (\mathcal{X}, \mathcal{E})$ and $\boldsymbol{X} \subset \mathcal{X}$. Define the *induced subgraph* $\mathcal{K}[\boldsymbol{X}]$ to be the graph $(\boldsymbol{X}, \mathcal{E}')$ where $\mathcal{E}'$ are all edges $X \rightleftharpoons Y$ such that $X, Y \in \boldsymbol{X}$.
- **Complete Subgraph**. A subgraph over $\boldsymbol{X}$ is *complete* if every two nodes in $\boldsymbol{X}$ are connected by some edge. The set $\boldsymbol{X}$ is often called a *clique*; we say that a clique $\boldsymbol{X}$ is *maximal* if for any superset of nodes $\boldsymbol{Y} \supset \boldsymbol{X}$, $\boldsymbol{Y}$ is not a clique.
- **Upward Closure**. We say that a subset of nodes $\boldsymbol{X} \in \mathcal{X}$ is *upwardly closed* in $\mathcal{K}$ if $\forall X \in \boldsymbol{X}$, we have that $\text{Boundary}_X \subset \boldsymbol{X}$[158]. We define the *upward closure* of $\boldsymbol{X}$ to be the minimally upwardly closed subset $\boldsymbol{Y}$ that contains $\boldsymbol{X}$. We define the *upwardly closed subgraph* of $\boldsymbol{X}$, denoted $\mathcal{K}^+[\boldsymbol{X}]$, to be the induced subgraph over $\boldsymbol{Y}$, $\mathcal{K}[\boldsymbol{Y}]$.

Paths and Trails. Definitions for longer-range connections in graphs. We use the notation $X_i \rightleftharpoons X_{j+1}$ to denote that X_i and X_j are connected via some edge, whether directed (in any direction) or undirected.

- **Trail/Path**. We say that $X_1, \ldots, X_k$ form a *trail* in the graph $\mathcal{K} = (\mathcal{X}, \mathcal{E})$ if $\forall i = 1, \ldots, k-1$, we have that $X_i \rightleftharpoons X_{j+1}$. A *path* makes an additional restriction: either $X_i \to X_{i+1}$ or $X_i — X_{i+1}$.
- **Connected Graph**. A graph is *connected* if $\forall X_i, X_j$ there is a trail between X_i and X_j.
- **Cycle**. A *cycle* in $\mathcal{K}$ is a directed path $X_1, \ldots X_k$ where $X_1 = X_k$.
- **Loop**. A *loop* in $\mathcal{K}$ is a trail where $X_1 = X_k$. A graph is *singly connected* if it contains no loops. A node in a singly connected graph is called a *leaf* if it has exactly one adjacent node.
- **Polytree/Forest**. A singly connected graph is also called a *polytree*. A singly connected undirected graph is called a *forest*; if a forest is also connected, it is called a *tree*.
  - A directed graph is a forest if each node has at most one parent. A directed forest is a tree if it is also connected.
- **Chordal Graph**. Let $X_1 — X_2 — \cdots — X_k — X_1$ be a loop in the graph. A *chord* in the loop is an edge connecting X_i and X_j for two nonconsecutive nodes X_i, X_j. An undirected graph $\mathcal{H}$ is said to be *chordal* if any loop $X_1 — X_2 — \cdots — X_k — X_1$ for $k \geq 4$ has a chord.

[158] $\text{Boundary}_X \triangleq \text{Pa}_X \cup \text{Nb}_X$. For DAGs, this is simply X's parents, and for undirected graphs X's neighbors.

Probability. Some notational reminders for this book. Let Ω denote a space of possible outcomes, and let S denote a set of measurable **events** α, each of which are a subset of Ω.

A probability distribution P over (Ω, S) is a mapping from events in S to real values that satisfy:

- $P(\alpha) \geq 0$ *for all* $\alpha \in S$.
- $P(\Omega) = 1$.
- *If* $\alpha, \beta \in S$ *and* $\alpha \cap \beta = \varnothing$, *then* $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$.

Some useful independence properties:

$$\textbf{Symmetry}: \quad (X \perp Y \mid Z) \implies (Y \perp X \mid Z) \tag{668}$$

$$\textbf{Decomposition}: \quad (X \perp (Y, W) \mid Z) \implies (X \perp Y \mid Z) \tag{669}$$

$$\textbf{Weak Union}: \quad (X \perp (Y, W) \mid Z) \implies (X \perp Y \mid Z, W) \tag{670}$$

$$\textbf{Contraction}: \quad (X \perp W \mid Z, Y) \& (X \perp Y \mid Z) \implies (X \perp Y, W \mid Z) \tag{671}$$

My Proofs: Independence Properties

I'll be using the definition that $(X \perp Y \mid Z) \Leftrightarrow P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. Given this definition the proof for the symmetry property is trivial. In what follows, I'll assume the LHS of the given implication is true, and then show that the RHS must hold as well.

Decomposition:

$$P(X, Y \mid Z) = \sum_w P(X, Y, w \mid Z) = \sum_w P(X \mid Z)P(Y, w \mid Z) = P(X \mid Z)P(Y \mid Z) \quad \checkmark$$

Weak Union:

$$\begin{aligned}
P(X, Y \mid Z, W) &= \frac{P(X, Y, W \mid Z)}{P(W \mid Z)} && (672)\\
&= \frac{P(X \mid Z)P(Y, W \mid Z)}{P(W \mid Z)} && (673)\\
&= \frac{P(X \mid Z)P(W \mid Z)P(Y \mid Z, W)}{P(W \mid Z)} && (674)\\
&= P(X \mid Z, W)P(Y \mid Z, W) \quad \checkmark && (675)
\end{aligned}$$

Contraction:

$$\begin{aligned}
P(X, Y, W \mid Z) &= P(Y \mid Z)P(X, W \mid Z, Y) && (676)\\
&= P(Y \mid Z)P(X \mid Z, Y)P(W \mid Z, Y) && (677)\\
&= P(X \mid Z)\left[P(Y \mid Z)P(W \mid Z, Y)\right] && (678)\\
&= P(X \mid Z)P(Y, W \mid Z) \quad \checkmark && (679)
\end{aligned}$$

We now define what "positive distribution" means, and a useful property of such distributions.

A distribution P is said to be ***positive*** *if for all events $\alpha \in S$, such that $\alpha \neq \varnothing$, we have that $P(\alpha) > 0$.*

For positive distributions, and for mutually disjoint sets $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}, \boldsymbol{W}$, the **intersection** property also holds:

$$\textbf{Intersection}: \qquad (\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z}, \boldsymbol{W}) \& (\boldsymbol{X} \perp \boldsymbol{W} \mid \boldsymbol{Z}, \boldsymbol{Y}) \implies (\boldsymbol{X} \perp \boldsymbol{Y}, \boldsymbol{W} \mid \boldsymbol{Z}) \tag{680}$$

Appendix

Figured this would be a good place to put some of the definitions in the Appendix, too.

Information Theory (A.1).

- **Entropy**. Let $P(X)$ be a distribution over a random variable X. The *entropy* of X is defined as.

$$H_P(X) = \mathbb{E}_P\left[\lg \frac{1}{P(X)}\right] = \sum_x P(X = x) \lg \frac{1}{P(X = x)} \tag{681}$$

We treat $0 \log \frac{1}{0} = 0$

$$0 \leq H_P(X) \leq \lg |Val(X)| \tag{682}$$

  $H_p(X)$ is a lower bound for the expected number of bits required to encode instances sampled from $P(X)$. Another interpretation is that the entropy is a measure of our uncertainty about the value of X.
- **Conditional Entropy**. The *conditional entropy* of X given Y is

$$H_P(X \mid Y) = H_P(X, Y) - H_P(Y) = \mathbb{E}_P\left[\lg \frac{1}{P(X \mid Y)}\right] \tag{683}$$

$$H_P(X \mid Y) \leq H_P(X) \tag{684}$$

  which captures the additional cost (in bits) of encoding X when we're already encoding Y.
- **Mutual Information**. The *mutual information* between X and Y is

$$I_P(X; Y) = H_P(X) - H_P(X \mid Y) = \mathbb{E}_P\left[\lg \frac{P(X \mid Y)}{P(X)}\right] \tag{685}$$

  which captures how many bits we save (on average) in the encoding of X if we know the value Y.
- **Distance Metric**. A *distance metric* is any distance measure d evaluating the distance between two distributions that satisfies all of the following properties:
  - **Positivity**: $d(P, Q) \geq 0$ and $d(P, Q) = 0$ if and only if $P = Q$.

- **Symmetry**: $d(P, Q) = d(Q, P)$.
- **Triangle inequality**: For any three distributions P, Q, R, we have that

$$d(P, R) \leq d(P, Q) + d(Q, R) \tag{686}$$

- **Kullback-Liebler Divergence**. Let P and Q be two distributions over random variables $X_1, \ldots X_n$. The relative entropy, or KL-divergence, of P and Q is

$$D(P \| Q) = \mathbb{E}_P \left[\lg \frac{P(X_1, \ldots, X_n)}{Q(X_1, \ldots, X_n)} \right] \tag{687}$$

Note that this only satisfies the positivity property, and is thus not a true distance metric.

Algorithms and Algorithmic Complexity (A.3).

- **Decision Problems**. A decision problem Π is a task that accepts an input (instance) ω and decides whether it satisfies a certain condition or not. The **SAT** problem accepts a formula in propositional logic and decides whether there is an assignment to the variables in the formula such that it evaluates to true. **3-SAT** restricts this to accepting only formulas in conjunctive normal form (CNF), and further restricted s.t. each clause contains at most 3 literals.
- **P**. A decision problem is in the class $\mathcal{P}$ if there exists a deterministic algorithm that takes an instance ω and determines whether $\omega \in \mathcal{L}_\Pi$ (the set of instances for which a correct algorithm must return true), in polynomial time in the size of the input ω.
- **NP**. A *non-deterministic algorithm* takes the general form: (1) nondeterministically guess some assignment γ to the variables of ω, (2) deterministically verify whether γ satisfies the condition of the problem.The algorithm will repeat these steps until it produces a γ that satisfies the problem. A decision problem Π is in the class $\mathcal{NP}$ if there exists a nondeterministic algorithm that accepts ω if and only if $\omega \in \mathcal{L}_\Pi$, and if the verification stage can be executed in polynomial time in the length of ω.
- **NP-hard**. Π is $\mathcal{NP}$-hard if for every DP $\Pi' \in \mathcal{NP}$, there is a polynomial-time transformation of inputs such that an input for Π' belongs to $\mathcal{L}_{\Pi'}$ if and only if the transformed instance belongs to $\mathcal{L}_\Pi$. Note that $\mathcal{NP}$-hard is a superset of $\mathcal{NP}$.

The SAT problem is $\mathcal{NP}$-hard.

- **NP-complete**. A problem Π is said to be $\mathcal{NP}$-complete if it is both $\mathcal{NP}$-hard and in $\mathcal{NP}$.

Combinatorial Optimization and Search (A.4). Below, I'll outline some common search algorithms. These are designed to address the following task:

> *Given initial candidate solution σ_{cur}, a score function* `score`*, and a set of* ***search operators*** *$\mathcal{O}$, search for the optimal solution σ_{best} that maximizes the value of $score(\sigma_{best})$*

Greedy local search (Algorithm A.5)

Repeat the following until *didUpdate* evaluates to $false$ at the end of an iteration.

1. Initialize $\sigma_{best} := \sigma_{cur}$.
2. Set $didUpdate := false$.
3. For each operator $o \in \mathcal{O}$, do:
   (a) Let $\sigma_o := o(\sigma_{best})$.
   (b) If σ_o is legal solution, and $score(\sigma_o) > score(\sigma_{best})$, reassign $\sigma_{best} := \sigma_o$, and set $didUpdate := true$.
4. If $didUpdate == true$, go back to step 2. Otherwise terminate and return σ_{best}.

Beam search (Algorithm A.7)

We are given a **beam width** K. Initialize our *beam*, the set of at most K solutions we are currently tracking, to $\{\sigma_{cur}\}$. Repeat the following until termination[a]:

1. Initial the set of successor states $H := \varnothing$.
2. For each solution $\sigma \in Beam$, and each operator $o \in \mathcal{O}$, insert a candidate successor state $o(\sigma)$ into H.
3. Set $Beam := KBestScore(H)$[b].

Once termination is reached, return the best solution σ_{best} in $Beam$.

[a] Termination condition could be e.g. an upper bound on number of iterations or on the improvement achieved in the last iteration.

[b] Notice that this implies an underlying assumption of beam search: all successor states $\sigma \in H$ have scores greater than any of the states in the current beam. We always assume improvement.

Continuous Optimization (A.5).

- **Line Search**. Method for adaptively choosing the step size (learning rate) η at each training step. Assuming we are doing gradient ascent. We'd usually set the parameters θ at step $t+1$ to $\theta^{(t)} + \eta \nabla f(\theta^{(t)})$. Line search modifies this by instead defining the "line" $g(\eta)$ below, and searching for the optimal value of η along that line.

$$g(\eta) = \overrightarrow{\theta}^{(t)} + \eta \nabla f(\theta^{(t)}) \tag{688}$$

$$\eta^{(t)} = \arg\max_{\eta} [g(\eta)] \tag{689}$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \eta^{(t)} \nabla f(\theta^{(t)}) \tag{690}$$

At risk of stating the obvious, $g(\eta)$ is referred to as a "line" because it's a function of the form `mx + b`.

L-BFGS

Some notes from the textbook "Numerical Optimization" (chapters 8 and 9).

The BFGS Method (8.1). Begin the derivation by forming the following quadratic model of the objective function f at the current iterate[159] θ_t:

For $m_t(p)$, p denotes the deviation at step t from the current parameters θ_t.

$$m_t(p) = f_t + \nabla f_t^T p + \frac{1}{2} p^T B_t p \tag{691}$$

where B_t is an $n \times n$ symmetric p.d. matrix that will be revised/updated every iteration (it is *not* the Hessian!). The minimizer p_t of this function can be written explicitly

$$p_t = -B_t^{-1} \nabla f_t \tag{692}$$

$\frac{\partial}{\partial p} \frac{1}{2} p^T B_t p = B_t p$

is used as the search direction, and the new iterate is

$$\theta_{t+1} \leftarrow \theta_t + \alpha_t p_t \tag{693}$$

where the step length α_t is chosen to satisfy the **Wolfe conditions**[160]. This is basically Newton's method with line search, except that we're using the *approximate Hessian* B_t instead of the true Hessian. It would be nice if we could somehow avoid recomputing B_t at each step. One proposed method involves imposing conditions on B_{t+1} based on the previous step(s). Require that ∇m_{t+1} equal ∇f at the latest two iterates θ_t and θ_{t+1}. Formally, the two conditions can be written as

$$\nabla m_{t+1}(-\alpha_t p_t) = \nabla f_{t+1} - \alpha_t B_{t+1} p_t = \nabla f_t \tag{696}$$

$$\nabla m_{t+1}(0) = \nabla f_{t+1} \tag{697}$$

We can rearrange the first condition to obtain the **secant equation**:

$$H_{t+1} y_t = s_t \quad \text{where} \tag{698}$$

$$H_{t+1} \triangleq B_{t+1}^{-1} \tag{699}$$

$$s_t \triangleq \theta_{t+1} - \theta_t \tag{700}$$

$$y_t \triangleq \nabla f_{t+1} - \nabla f_t \tag{701}$$

which is true only if s_t and y_t satisfy the **curvature condition**, $s_t^T y_t > 0$. The curvature condition is guaranteed to hold if we impose the Wolfe conditions on the line search. As is, this

[159] Recall that an "iterate" is just some variable that gets iteratively computed/updated. Fancy people with fancy words.

[160] The Wolfe conditions are the following sufficient decrease and curvature conditions for line search:

$$f(\theta_t + \alpha_t p_t) \leq f(\theta_t) + c_1 \alpha_t \nabla f_t^T p_t \tag{694}$$

$$\nabla f(\theta_t + \alpha_t p_t)^T p_t \geq c_2 \nabla f_t^T p_t \tag{695}$$

for some constant $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$.

still has infinitely many solutions for H_{t+1}. To determine it uniquely, we impose the additional condition that H_{t+1} is the closest of all possible solutions to the current H_t:

$$\min_H ||H - H_t||_W \quad \text{s.t.} \quad H = H^T,\ Hy_t = s_t \tag{702}$$

where $||\cdot||_W$ is the *weighted Frobenius norm*[161], and W can be any matrix satisfying $Ws_t = y_t$. For concreteness, assume that $W = \widetilde{G}_t$, where

$$\widetilde{G}_t = \int_0^1 \nabla^2 f(\theta_t + \tau\alpha_t p_t)\mathrm{d}\tau \tag{705}$$

The unique solution to H_{t+1} is given by

$$\textbf{(BFGS)} \qquad H_{t+1} = (I - \rho_t s_t y_t^T)H_t(I - \rho_t y_t s_t^T) + \rho_t s_t s_t^T \tag{8.16}$$

where $\rho_t = 1/(y_t^T s_t)$. The BFGS is summarized in algorithm 8.1 below.

Algorithm 8.1 (BFGS Method). Given starting point θ_0, convergence tolerance $\epsilon > 0$, and inverse Hessian approximation H_0. Initialize $t = 0$. While $||\nabla f_t|| > \epsilon$ do:

1. Compute search direction $p_t = -H_t \nabla f_t$.
2. Set $\theta_{t+1} = \theta_t + \alpha_t p_t$, where α_t is computed via line search to satisfy the Wolfe conditions.
3. Define $s_t = \theta_{t+1} - \theta_t$ and $y_t = \nabla f_{t+1} - \nabla f_t$.
4. Compute H_t by means of equation 8.16.
5. Increment $t \mathrel{+}= 1$ and go back to step 1.

161

$$||H||_W \triangleq ||W^{1/2} H W^{1/2}||_F \tag{703}$$

$$||C||_F^2 \triangleq \sum_{i,j} c_{ij}^2 \tag{704}$$

L-BFGS. Modifies BFGS to store a modified version of H_t implicitly, by storing some number m of vector pairs $\{s_i, y_i\}$, corresponding to the m most recent time steps. We use a recursive procedure to compute $H_t \nabla f_t$ given the set of vectors.

Algorithm 9.1 (L-BFGS two-loop recursion) Subroutine of L-BFGS for computing $H_t \nabla f_t$. We're given the current value of ∇f_t, and we initialize q to this value.

1. For i in the range $[t-1, t-m]$, compute

$$\alpha_i \leftarrow \rho_i s_i^T q \tag{706}$$

$$q \leftarrow q - \alpha_i y_i \tag{707}$$

2. Set $r \to H_t^0 q$.
3. For i in the range $[t-m, t-1]$, compute

$$\beta \leftarrow \rho_i y_i^T r \tag{708}$$

$$r \leftarrow r + s_i(\alpha_i - \beta) \tag{709}$$

4. Return result $H_t \nabla f_t = r$.

Algorithm 9.2 (L-BFGS). Given starting point θ_0, integer $m > 0$, and initial $t = 0$. Repeat the following until convergence.

1. Choose H_t^0. A popular choice is $H_t^0 := \gamma_t I$, where

$$\gamma_t \triangleq \frac{s_{t-1}^T y_{t-1}}{y_{t-1}^T y_{t-1}}$$

2. Compute $p_t \leftarrow -H_t \nabla f_t$ from Algorithm 9.1.
3. Compute $\theta_{t+1} \leftarrow \theta_t + \alpha_t p_t$, where α_t is chosen to satisfy the Wolfe conditions.
4. if $t > m$, discard $\{s_{t-m}, y_{t-m}\}$. Compute and save s_t and y_t.
5. Increment $t \mathrel{+}= 1$ and go back to step 1.

Exercises

Going through all the problems with a star for review.

Exercise 2.4

Let $\alpha \in S$ be an event s.t. $P(\alpha) > 0$. Show that $P(\cdot \mid \alpha)$ satisfies the properties of a valid probability distribution.

- Show $P(\beta \mid \alpha) \geq 0$ for all $\beta \in S$. By definition,

$$P(\beta \mid \alpha) = \frac{1}{P(\alpha)} P(\alpha \cap \beta) \tag{710}$$

  and since the full joint $P \geq 0$ and since $P(\alpha) > 0$, we have the desired result.
- Show $P(\Omega_\alpha) = 1$. Again, using just the definitions,

$$\begin{aligned} P(\Omega_\alpha) &= \sum_{\beta \in S} P(\beta \mid \alpha) && (711) \\ &= \frac{1}{P(\alpha)} \sum_{\beta \in S} P(\alpha \cap \beta) && (712) \\ &= \frac{1}{P(\alpha)} \left(\sum_{\beta \in \alpha} P(\beta) + \sum_{\gamma \notin \alpha} P(\varnothing) \right) && (713) \\ &= \frac{1}{P(\alpha)} \left(P(\alpha) + 0 \right) && (714) \\ &= 1 && (715) \end{aligned}$$

- Show, for any $\beta, \gamma \in S$, where $\beta \cap \gamma = \varnothing$, that $P(\beta \cup \gamma \mid \alpha) = P(\beta \mid \alpha) + P(\gamma \mid \alpha)$.

$$\begin{aligned} P(\beta \cup \gamma \mid \alpha) &= \frac{1}{P(\alpha)} P((\beta \cup \gamma) \cap \alpha) && (716) \\ &= \frac{1}{P(\alpha)} P((\beta \cap \alpha) \cup (\gamma \cap \alpha)) && (717) \\ &= \frac{1}{P(\alpha)} \left(P(\beta \cap \alpha) + P(\gamma \cap \alpha) \right) && (718) \\ &= P(\beta \mid \alpha) + P(\gamma \mid \alpha) && (719) \end{aligned}$$

Exercise 2.16: Jensen's Inequality

Let f be a concave function and P a distribution over a random variable X. Then

$$\mathbb{E}_P\left[f(X)\right] \leq f(\mathbb{E}_P\left[X\right]) \tag{720}$$

Use this inequality to prove the following 3 properties.

- $H_P(X) \leq \log|Val(X)|$. Let $f(u) := \lg(u)$ be our concave function.

$$H_P(X) \triangleq \mathbb{E}_P\left[\lg\frac{1}{P(X)}\right] = \mathbb{E}_P\left[f(u)\right] \tag{721}$$

$$\leq f(\mathbb{E}_p\left[u\right]) = f\left(\sum_x u(x)P(x)\right) = f(|Val(X)|) = \lg|Val(X)| \tag{722}$$

$$\tag{723}$$

- $H_P(X) \geq 0$.

$$-H_P(X) = -\mathbb{E}_P\left[\lg\frac{1}{P(X)}\right] \tag{724}$$

$$= \mathbb{E}_P\left[\lg P(X)\right] \leq 0 \tag{725}$$

  since $0 \leq P(X = x) \leq 1\ \forall x$ (any term in the expectation where $P(X = x) = 0$ is equal to 0, by definition).
- $D(P||Q) \geq 0$. Use the same idea as in the first proof, but let $u(x) = Q(x)/P(x)$.

$$D(P||Q) \triangleq \mathbb{E}_P\left[\lg\left(P(X)/Q(X)\right)\right] = -\mathbb{E}_p\left[f(u)\right] \tag{726}$$

$$-\mathbb{E}_p\left[f(u)\right] \geq f\left(\sum_x u(x)P(x)\right) = f(1) = 0 \tag{727}$$

The Bayesian Network Representation (Ch. 3)

Koller and Friedman (2009). The Bayesian Network Representation.
Probabilistic Graphical Models: Principles and Techniques.

Goal: *represent a joint distribution P over some set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$.* Consider the case where each X_i is binary-valued. A *single* joint distribution requires access to the probability for each of the 2^n possible assignments for $\mathcal{X}$. The set of all such possible joint distributions,

$$\{(p_1, \ldots, p_{2^n}) \in \mathbb{R}^{2^n} : \sum_{i=1}^{2^n} p_i = 1\} \tag{728}$$

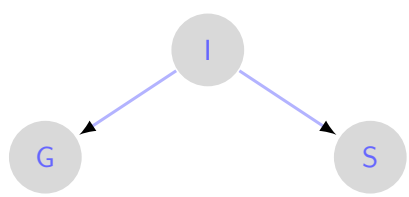
Understanding the exponential blowup.

is a $2^n - 1$ dimensional subspace of $\mathbb{R}^{2^n}$. Note that each p_i represents the probability for a unique instantiation of $\mathcal{X}$. Furthermore, in the general case, knowing p_i tells you nearly nothing about $p_{j \neq i}$ – i.e. you require an instantiation of $2^n - 1$ independent parameters to specify a given joint distribution.

But it would be foolish to parameterize any joint distribution in this way, since we can often take advantage of independencies. Consider the case where each X_i gives the outcome (H or T) of coin i being tossed. Then our distribution satisfies $(\boldsymbol{X} \perp \boldsymbol{Y})$ for any disjoint subsets fo the variables $\boldsymbol{X}$ and $\boldsymbol{Y}$. Let θ_i denote the probability that coin i lands heads. The key observation is that you only need each of the n θ_i to specify a unique joint distribution over $\mathcal{X}$, reducing the $2^n - 1$ dimensional subspace to an n dimensional manifold in $\mathbb{R}^{2^n}$[162].

Taking advantage of independencies.

The Naive Bayes Model. Say we want to determine the intelligence of an applicant based on their grade G in some course, and their score S on the SAT. A naive bayes model can be illustrated as below.



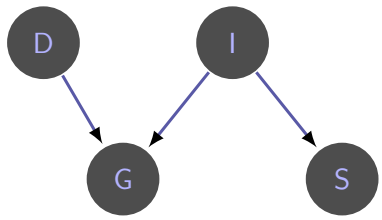
This encodes our general assumption that[163] $P \models (S \perp G \mid I)$.

[162] **TODO**: Formally define this manifold using the same notation as in 728. Edit: Not sure how to actually write it out, but intuitively it's because each of the 2^n p_i values, when going from the general case to the case of i.i.d, become *functions* of the n θ_i values. Whereas before they were independent free parameters.

[163] We say that an event α is independent of event β in P with the notation $P \models (\alpha \perp \beta)$. (Def 2.2, pg 23 of book)

In general, a naive bayes model assumes that instances fall into one of a number of mutually exclusive and exhaustive *classes*, defined as the set of values that the top variable in the graph can take on[164]. The model also includes some number of *features* $X_1, \ldots X_k$, whose values are typically observed. The **naive Bayes assumption** is that the features are conditionally independent given the instance's class.

Bayesian Networks. A Bayesian network $\mathcal{B}$ is defined by a network structure together with its set of CPDs. **Causal reasoning** (or prediction) refers to computing the downstream effects of various factors (such as intelligence). **Evidential reasoning** (or explanation) is the reverse case, where we reason from effects to causes. Finally, **intercausal reasoning** (or explaining away) is when different causes of the same effect can interact. For our student example, we could be trying to determine $\Pr[I \mid G]$, the intelligence of an a student given his/her grade in a class. In addition to intelligence being a cause for the grade, we could have another causal variable D for the difficulty of the class:



An example of intercausal reasoning would be observing D, so that we now we want $\Pr[I \mid G, d]$; the diffulty of the course can help *explain away* good/bad grades, thus changing our value for the probability of intelligence based on the grade alone.

A **Bayesian network structure** $\mathcal{G}$ is a DAG whose nodes represent RVs $X_1, \ldots X_n$. Let $Pa^{\mathcal{G}}_{X_i}$ denote the parents of X_i in $\mathcal{G}$, and NonDesc_{X_i} denotes the variables that are NOT descendants of X_i. Then $\mathcal{G}$ encodes the following set of ***local independencies***,

$$\mathcal{I}_\ell(\mathcal{G}) \triangleq \{\forall X_i : (X_i \perp \text{NonDesc}_{X_i} \mid Pa^{G}_{X_i})\} \tag{729}$$

[164]For the intelligence example, the classes are high intelligence and low intelligence.

Graphs and Distributions. Here we see that a distribution P satisfies the local independencies associated with a graph $\mathcal{G}$ iff P is representable as a set of CPDs associated with $\mathcal{G}$.

- **Local independencies**. Let P be a distribution over $\mathcal{X}$. We define $\mathcal{I}(P)$ to be the set of **independence assertions** of the form $(\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})$ that hold in P. The statement "P satisfies the local independencies associated with $\mathcal{G}$" can thus be succinctly written:

$$\mathcal{I}_\ell(\mathcal{G}) \subseteq \mathcal{I}(P) \tag{730}$$

  and we'd say that $\mathcal{G}$ is an **I-map** (independency map) for P.
- **I-maps**. More generally, let $\mathcal{K}$ be *any* graph object associated with a set of independencies $\mathcal{I}(\mathcal{K})$. We say that $\mathcal{K}$ is an **I-map** for a set of independencies $\mathcal{I}$ if $\mathcal{I}(\mathcal{K}) \subseteq \mathcal{I}$. Note that the complete graph (every two nodes connected) is an I-map for any distribution.
- **Factorization**. Let $\mathcal{G}$ be a BN graph over $X_1, \ldots, X_n$. We say that a distribution P over the same space **factorizes** according to $\mathcal{G}$ if P can be expressed as

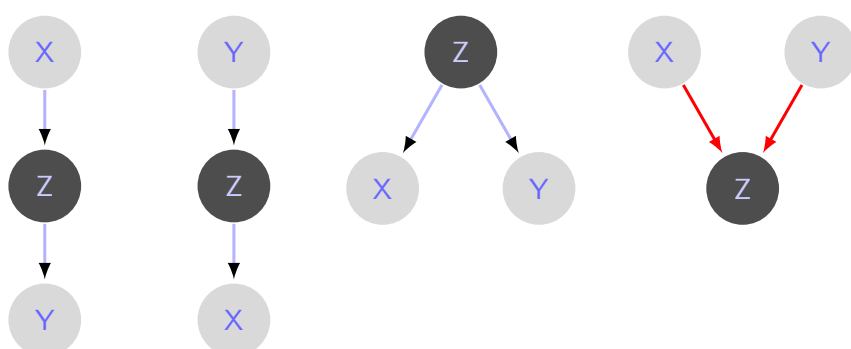
$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid Pa_{X_i}^{\mathcal{G}}) \tag{731}$$

  For BNs, the following equivalence holds:

$$\mathcal{G} \text{ is an I-map for } P \iff P \text{ factorizes according to } \mathcal{G}$$

D-separation. We want to understand when we can *guarantee* that an independence $(\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})$ holds in a distribution associated with a BN structure $\mathcal{G}$. Consider a 3-node network consisting of X, Y, and Z, where X and Y are not directly connected. There are four such cases, which I've drawn below.

The fourth trail is called a *v-structure*.



From left-to-right: indirect causal effect, indirect evidential effect, common cause, common effect. The first 3 satisfy $(X \perp Y \mid Z)$, but the 4th does not. Another way of saying this is that the first 3 trails are **active**[165] IFF Z is not observed, while the 4th trail is active IFF Z (or a descendent of Z) is observed.

[165] When influence can flow from X to Y via Z, we say that the trail $X \rightleftharpoons Y \rightleftharpoons Z$ is *active*.

General case:

> *Let $\mathcal{G}$ be a BN structure, and $X \rightleftharpoons \cdots \rightleftharpoons X_n$ a trail in $\mathcal{G}$. Let $\boldsymbol{Z}$ be a subset of observed variables. The trail is* ***active*** *given $\boldsymbol{Z}$ if*
>
> - *Any v-structure $X_{i-1} \to X_i \leftarrow X_{i+1}$ has X_i or one of its descendants in $\boldsymbol{Z}$.*
> - *No other node along the trail is in $\boldsymbol{Z}$.*

D-separation: (Directed separation)

> *Let $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$ be three sets of nodes in $\mathcal{G}$. We say that $\boldsymbol{X}$ and $\boldsymbol{Y}$ are* ***d-separated*** *in $\mathcal{G}$, denoted $\text{d-sep}_{\mathcal{G}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z})$, if there is no active trail between any node $X \in \boldsymbol{X}$ and $Y \in \boldsymbol{Y}$ given $\boldsymbol{Z}$. We use $\mathcal{I}(\mathcal{G})$ to denote the set of independencies that correspond to d-separation,*
>
> $$\mathcal{I}(\mathcal{G}) = \{(\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z}) : \text{d-sep}_{\mathcal{G}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z})\} \tag{732}$$
>
> *also called the set of* ***global Markov independencies****.*

Soundness and Completeness of d-separation as a method for determining independence.

- **Soundness** (Thm 3.3). If a distribution P factorizes according to $\mathcal{G}$, then $\mathcal{I}(\mathcal{G}) \subseteq \mathcal{I}(P)$.
- **Completeness**. For any distribution P that factorizes over $\mathcal{G}$, we have that P is **faithful**[166] to $\mathcal{G}$:

$$X \perp Y \mid \boldsymbol{Z} \in \mathcal{I}(P) \implies \text{d-sep}_{\mathcal{G}}(X; Y \mid \boldsymbol{Z}) \tag{733}$$

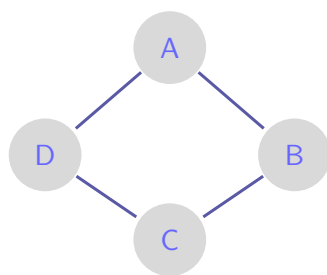
To see the detailed algorithm for finding nodes reachable from X given $\boldsymbol{Z}$ via active trails, see Algorithm 3.1 on pg. 75 of the book.

[166] P is faithful to $\mathcal{G}$ if, whenever $X \perp Y \mid \boldsymbol{Z} \in \mathcal{I}(P)$, then $\text{d-sep}_{\mathcal{G}}(X; Y \mid \boldsymbol{Z})$.

Undirected Graphical Models (Ch. 4)

Koller and Friedman (2009). Undirected Graphical Models.
Probabilistic Graphical Models: Principles and Techniques.

The Misconception Example. Consider a scenario where we have four students who get together in pairs to work on homework for a class. Only the following pairs meet: (Alice, Bob), (Bob, Charles), (Charles, Debbie), (Debbie, Alice). The professor misspoke in class, giving rise to a possible misconception among the students. We have four binary random variables, $\{A, B, C, D\}$, representing whether the student has the misconception (1) or not (0) [167]. Intuitively, we want to model a distribution that satisfies $(A \perp C \mid \{B, D\})$, and $(B \perp D \mid \{A, C\})$, but no other independencies[168]. Note that the interactions between variables seem symmetric here – students influence each other (out of the ones they have a pair with).



The nodes in the graph of a **Markov network** represent the variables, and the edges correspond to a notion of direct probabilistic interaction between the neighboring variables – an interaction that is not mediated by any other variable in the network. So, how should we parameterize our network? We want to capture the **affinities** between the related variables (e.g. Alice and Bob are more likely to agree than disagree).

We restrict our attention to nonnegative factors.

> *Let $\boldsymbol{D}$ be a set of random variables. We define a **factor** ϕ to be a function from $Val(\boldsymbol{D})$ to $\mathbb{R}$. A factor is nonnegative if all its entries are nonnegative. $\boldsymbol{D}$ is called the **scope** of the factor, denoted $Scope[\phi]$.*

The factors need not be normalized. Therefore, to interpret probabilities over factors, we must

[167] A student might not have the misconception if e.g. they went home and figured out the problem via reading the textbook instead.
[168] These independences cannot be naturally captured in a Bayesian (i.e. directed) network.

normalize it with what we'll call the **partition function**, Z:

$$\Pr[a,b,c,d] = \frac{1}{Z}\phi_1(a,b)\cdot\phi_2(b,c)\cdot\phi_3(c,d)\cdot\phi_4(d,a) \tag{734}$$

$$Z = \sum_{a,b,c,d}\phi_1(a,b)\cdot\phi_2(b,c)\cdot\phi_3(c,d)\cdot\phi_4(d,a) \tag{735}$$

Parameterization. Associating the graph structure with a set of parameters. We parameterize undirected graphs by associating a set of factors with it. First, we introduce the definition of **factor product**:

> *Let $\boldsymbol{X}$, $\boldsymbol{Y}$, and $\boldsymbol{Z}$ be three disjoint sets of variables, and let $\phi_1(\boldsymbol{X},\boldsymbol{Y})$ and $\phi_2(\boldsymbol{Y},\boldsymbol{Z})$ be two factors. We define the factor product $\phi_1\times\phi_2$ to be a factor $\psi : Val(\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z})\mapsto\mathbb{R}$ as follows:*
>
> $$\psi(\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}) = \phi_1(\boldsymbol{X},\boldsymbol{Y})\cdot\phi_2(\boldsymbol{Y},\boldsymbol{Z}) \tag{736}$$

We use this to define an undirected parameterization of a distribution:

> *A distribution $P_{\boldsymbol{\Phi}}$ is a **Gibbs distribution** parameterized by a set of factors $\boldsymbol{\Phi} = \{\phi_1(\boldsymbol{D}_1,\ldots,\phi_K(\boldsymbol{D}_K))\}$ if it is defined as follows:*
>
> $$P_{\boldsymbol{\Phi}} = \frac{1}{Z}\widetilde{P}_{\boldsymbol{\Phi}}(X_1,\ldots,X_n) \tag{737}$$
>
> $$\widetilde{P}_{\boldsymbol{\Phi}}(X_1,\ldots,X_n) = \phi_1(\boldsymbol{D}_1)\times\phi_2(\boldsymbol{D}_2)\times\cdots\times\phi_K(\boldsymbol{D}_K) \tag{738}$$

where the authors (Koller and Friedman) have made a point to emphasize: **A factor is only one contribution to the overall joint distribution. The distribution as a whole has to take into consideration the contributions from all of the factors involved.** Now, we relate the parameterization of a Gibbs distribution to a graph structure.

> *A distribution $P_{\boldsymbol{\Phi}}$ with $\boldsymbol{\Phi} = \{\phi_1(\boldsymbol{D}_1,\ldots,\phi_K(\boldsymbol{D}_K))\}$ **factorizes** over a Markov network $\mathcal{H}$ if each $\boldsymbol{D}_k (k=1,\ldots,K)$ is a complete subgraph[169] of $\mathcal{H}$.*

The factors that parameterize a Markov network are often called **clique potentials**. Although it can be used without loss of generality, the parameterization using maximal clique potentials generally obscures structure that is present in the original set of factors. Below are some useful definitions that we will use often.

[169] A subgraph is complete if every two nodes in the subgraph are connected by some edge. The set of nodes in such a subgraph is often called a **clique**. A clique $\boldsymbol{X}$ is maximal if for any superset of nodes $\boldsymbol{Y}\supset\boldsymbol{X}$, $\boldsymbol{Y}$ is *not* a clique.

Factor reduction:

Let $\phi(\boldsymbol{Y})$ be a factor, and $\boldsymbol{U} = \boldsymbol{u}$ an assignment for $\boldsymbol{U} \subseteq \boldsymbol{Y}$. Define the ***reduction*** *of the factor ϕ to the context $\boldsymbol{U} = \boldsymbol{u}$, denoted $\phi[\boldsymbol{u}]$, to be a factor over the scope $\boldsymbol{Y}' = \boldsymbol{Y} - \boldsymbol{U}$, such that*

$$\phi[\boldsymbol{u}](\boldsymbol{y}') = \phi(\boldsymbol{y}', \boldsymbol{u}) \tag{739}$$

For $\boldsymbol{U} \nsubseteq \boldsymbol{Y}$, define $\phi[\boldsymbol{u}]$ only for the assignments in $\boldsymbol{u}$ to the variables in $\boldsymbol{U}' = \boldsymbol{U} \cap \boldsymbol{Y}$.

Reduced Gibbs distribution:

Let $P_{\boldsymbol{\Phi}}(\boldsymbol{X})$ be a Gibbs distribution parameterized by $\boldsymbol{\Phi} = \{\phi_1, \ldots, \phi_K\}$ and let $\boldsymbol{u}$ be a context. The ***reduced Gibbs distribution*** *$P_{\boldsymbol{\Phi}}[\boldsymbol{u}]$ is the Gibbs distribution defined by the set of factors $\boldsymbol{\Phi}[\boldsymbol{u}] = \{\phi_1[\boldsymbol{u}], \ldots, \phi_K[\boldsymbol{u}]\}$. More formally:*

$$P_{\boldsymbol{\Phi}}[\boldsymbol{u}] = P_{\boldsymbol{\Phi}}(\boldsymbol{W} \mid \boldsymbol{u}) \quad \textit{where} \quad \boldsymbol{W} = \boldsymbol{X} - \boldsymbol{U} \tag{740}$$

Reduced Markov Network:

Let $\mathcal{H}$ be a Markov network over $\boldsymbol{X}$ and $\boldsymbol{U} = \boldsymbol{u}$ a context. The ***reduced Markov network*** *$\mathcal{H}[\boldsymbol{u}]$ is a Markov network over the nodes $\boldsymbol{W} = \boldsymbol{X} - \boldsymbol{U}$, where we have an edge X—Y if there's an edge X—Y in $\mathcal{H}$.*

Note that if a Gibbs distribution $P_{\Phi}(\boldsymbol{X})$ factorizes over $\mathcal{H}$, then $P_{\Phi}[\boldsymbol{u}]$ factorizes over $\mathcal{H}[\boldsymbol{u}]$.

Markov Network Independencies. A formal presentation of the undirected graph as a representation of independence assertions.

- **Active Path**. Let $\mathcal{H}$ be a Markov network structure, and let X_1—$\cdots$—X_k be a path in $\mathcal{H}$. Let $\boldsymbol{Z} \subseteq \mathcal{X}$ be a set of *observed variables*. The path is *active* given $\boldsymbol{Z}$ if none of the X_i's is in $\boldsymbol{Z}$.
- **Separation**. A set of nodes $\boldsymbol{Z}$ *separates* $\boldsymbol{X}$ and $\boldsymbol{Y}$ in $\mathcal{H}$, denoted $\text{sep}_{\mathcal{H}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z})$, if there is no active path between any node $X \in \boldsymbol{X}$ and $Y \in \boldsymbol{Y}$ given $\boldsymbol{Z}$.
- **Global Independencies**. The *global independencies* associated with $\mathcal{H}$ are defined as

$$\mathcal{I}(\mathcal{H}) = \{(\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z}) : \text{sep}_{\mathcal{H}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z})\} \tag{741}$$

  This is the *separation criterion*. Note that the definition of separation is monotonic in $\boldsymbol{Z}$: if it holds for $\boldsymbol{Z}$, then it holds for any $\boldsymbol{Z}' \supset \boldsymbol{Z}$ as well[170].

[170] Which means that Markov networks are fundamentally incapable of representing nonmonotonic independence relations!

- **Soundness** of the separation criterion for detecting independence properties in distributions over $\mathcal{H}$. In other words, we want to prove that

$$(P \text{ factorizes over } \mathcal{H}) \implies [\text{sep}_{\mathcal{H}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z}) \implies P \models (\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})]$$

where the portion in brackets can equivalently be said as "$\mathcal{H}$ is an I-map for P".

Proof

- Consider the case where $\boldsymbol{X} \cup \boldsymbol{Y} \cup \boldsymbol{Z} = \mathcal{X}$.
- Then, any clique in $\mathcal{X}$ is fully contained in either $\boldsymbol{X} \cup \boldsymbol{Z}$ or $\boldsymbol{Y} \cup \boldsymbol{Z}$. In other words,

$$P(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}) = \frac{1}{Z} f(\boldsymbol{X}, \boldsymbol{Z}) g(\boldsymbol{Y}, \boldsymbol{Z}) \tag{742}$$

$$\text{which implies} \quad P(\boldsymbol{X}, \boldsymbol{Y} \mid \boldsymbol{Z}) = \frac{f(\boldsymbol{X}, \boldsymbol{Z}) g(\boldsymbol{Y}, \boldsymbol{Z})}{\sum_{x,y} f(x, \boldsymbol{Z}) g(y, \boldsymbol{Z})} \tag{743}$$

- We can prove that $P \models (\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})$ by showing that $P(\boldsymbol{X}, \boldsymbol{Y} \mid \boldsymbol{Z}) = P(\boldsymbol{X} \mid \boldsymbol{Z}) P(\boldsymbol{Y} \mid \boldsymbol{Z})$.

$$P(\boldsymbol{X}, \boldsymbol{Y} \mid \boldsymbol{Z}) = \frac{f(\boldsymbol{X}, \boldsymbol{Z}) g(\boldsymbol{Y}, \boldsymbol{Z})}{P(\boldsymbol{Z})} \tag{744}$$

$$= \frac{f(\boldsymbol{X}, \boldsymbol{Z}) g(\boldsymbol{Y}, \boldsymbol{Z})}{P(\boldsymbol{Z})} \frac{P(\boldsymbol{Z})}{P(\boldsymbol{Z})} \tag{745}$$

$$= \frac{f(\boldsymbol{X}, \boldsymbol{Z}) g(\boldsymbol{Y}, \boldsymbol{Z})}{P(\boldsymbol{Z})} \frac{\sum_x f(x, \boldsymbol{Z}) \sum_y g(y, \boldsymbol{Z})}{P(\boldsymbol{Z})} \tag{746}$$

$$= \frac{f(\boldsymbol{X}, \boldsymbol{Z}) \sum_y g(y, \boldsymbol{Z})}{P(\boldsymbol{Z})} \frac{g(\boldsymbol{Y}, \boldsymbol{Z}) \sum_x f(x, \boldsymbol{Z})}{P(\boldsymbol{Z})} \tag{747}$$

$$= P(\boldsymbol{X} \mid \boldsymbol{Z}) P(\boldsymbol{Y} \mid \boldsymbol{Z}) \tag{748}$$

- For the general case where $\boldsymbol{X} \cup \boldsymbol{Y} \cup \boldsymbol{Z} = \mathcal{X}$, let $\boldsymbol{U} = \mathcal{X} - (\boldsymbol{X} \cup \boldsymbol{Y} \cup \boldsymbol{Z})$. Since we know that $\text{sep}_{\mathcal{H}}(\boldsymbol{X}; \boldsymbol{Y} \mid \boldsymbol{Z})$, we can partition $\boldsymbol{U}$ into two disjoint sets $\boldsymbol{U}_1$ and $\boldsymbol{U}_2$ such that $\text{sep}_{\mathcal{H}}(\boldsymbol{X} \cup \boldsymbol{U}_1; \boldsymbol{Y} \cup \boldsymbol{U}_2 \mid \boldsymbol{Z})$. Combining the previous result with the decomposition property[a] give us the desired result that $P \models (\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})$.

[a] The decomposition property:

$$(\boldsymbol{X} \perp (\boldsymbol{Y}, \boldsymbol{W}) \mid \boldsymbol{Z}) \implies (\boldsymbol{X} \perp \boldsymbol{Y} \mid \boldsymbol{Z})$$

Pairwise Independencies:

Let $\mathcal{H}$ be a Markov network. We define the ***pairwise independencies*** *assoc. with $\mathcal{H}$:*

$$\mathcal{I}_p(\mathcal{H}) = \{(X \perp Y \mid \mathcal{X} - \{X, Y\}) \ : \ X — Y \notin \mathcal{H}\} \tag{749}$$

which just says "X is indep. of Y given everything else if there's no edge between X and Y."

Local Independencies:

> *For a given graph $\mathcal{H}$, define the* ***Markov blanket*** *of X in $\mathcal{H}$, denoted $\mathrm{MB}_{\mathcal{H}}(X)$, to be the neighbors of X in $\mathcal{H}$. We define the* ***local independencies*** *associated with $\mathcal{H}$:*
>
> $$\mathcal{I}_\ell(\mathcal{H}) = \{(X \perp \mathcal{X} - \{X\} - \mathrm{MB}_{\mathcal{H}}(X) \mid \mathrm{MB}_{\mathcal{H}}(X)) \ : \ X \in \mathcal{X}\} \tag{750}$$
>
> which just says "X is indep. of the rest of the nodes in the graph given its immediate neighbors."

Log-Linear Models. Certain patterns involving particular values of variables for a given factor can often be more easily seen by converting factors into log-space. More precisely, we can rewrite a factor $\phi(\boldsymbol{D})$ as

$$\phi(\boldsymbol{D}) = e^{-\epsilon(\boldsymbol{D})} \tag{751}$$

$$\epsilon(\boldsymbol{D}) \triangleq -\ln \phi(\boldsymbol{D}) \tag{752}$$

$$\Pr[X_1, \ldots, X_n] \propto e^{-\sum \epsilon_i(\boldsymbol{D}_i)} \tag{753}$$

where $\epsilon(\boldsymbol{D})$ is often called an **energy function**. Note how ϵ can take on any value along the real line (i.e. removes our nonnegativity constraint)[171]. Also note that as the ϵ summation approaches 0, the probability approaches one.

This motivates introducing the notion of a **feature**, which is just a factor without the nonnegativity requirement. A popular type of feature is the **indicator feature** that takes on value 1 for some values $\boldsymbol{y} \in Val(\boldsymbol{D})$ and 0 otherwise. We can now provide a more general definition for our notion of log-linear models:

> *A distribution P is a* ***log-linear model*** *over a Markov network $\mathcal{H}$ if it is associated with:*
>
> - *A set of features $\mathcal{F} = \{f_1(\boldsymbol{D}_1), \ldots, f_k(\boldsymbol{D}_k)\}$ where each $\boldsymbol{D}_i$ is a complete subgraph (i.e. a clique)in $\mathcal{H}$.*
> - *A set of weights $w_1, \ldots, w_k$.*
>
> *such that*
>
> $$Pr[X_1, \ldots, X_n] = \frac{1}{Z} \exp\left[-\sum_{i=1}^{k} w_i f_i(\boldsymbol{D}_i)\right] \tag{754}$$

The log-linear model provides a much more compact representation for many distributions, especially in situations where variables have large domains (such as text).

[171] We seem to be implicitly assuming that the original factors are all *positive* (not just non-negative).

Box 4.C – Concept: Ising Models and Boltzmann Machines

The **Ising model**: Each atom is modeled as a binary RV $X_i \in \{+1, -1\}$ denoting its spin. Each pair of neighboring atoms is associated with energy function $\epsilon_{i,j}(x_i, x_j) = w_{i,j} x_i x_j$. We also have individual energy functions $u_i x_i$ for each atom. This defines our distribution:

$$P(\xi) = \frac{1}{Z} \exp\Big(- \sum_{i<j} w_{i,j} x_i x_j - \sum_i u_i x_i \Big) \tag{755}$$

The **Boltzmann distribution**: Now the variables are $X_i \in \{0, 1\}$. The distribution of each X_i given its neighbors is

$$P(x_i^1 \mid Nb(X_i)) = \text{sigmoid}(z) \tag{756}$$

$$z = -(\sum_j w_{i,j} x_j) - u_i \tag{757}$$

Box 4.D – Concept: Metric MRFs

Consider the pairwise graph $X_1, \ldots, X_n$ in the context of sequence labeling. We want to assign each X_i a label. We also want adjacent nodes to prefer being similar to each other. We usually use the MAP objective, so our goal will be to minimize the total energy over the parameters (which are given by the individual energy functions ϵ_i).

$$E(x_1, \ldots, x_n) = \sum_i \epsilon_i(x_i) + \sum_{(i,j) \in \mathcal{E}} \epsilon_{i,j}(x_i, x_j) \tag{758}$$

The simplest place to start for preferring neighboring labels to take on similar values is to define $\epsilon_{i,j}$ to have low energy when $x_i = x_j$ and some positive $\lambda_{i,j}$ otherwise. We want to have finer granularity for our similarities between labels. To do this, we introduce the definition of a **metric**: a function $\mu : \mathcal{V} \times \mathcal{V} \mapsto [0, \infty)$ that satisfies

$$\text{[reflexivity]} \qquad \mu(v_k, v_l) = 0 \quad \text{IFF} \quad k = l \tag{759}$$

$$\text{[symmetry]} \qquad \mu(v_k, v_l) = \mu(v_l, v_k) \tag{760}$$

$$\text{[triangle inequality]} \qquad \mu(v_k, v_l) + \mu(v_l, v_m) \geq \mu(v_k, v_m) \tag{761}$$

and we can now let $\epsilon_{i,j}(v_k, v_l) := \mu(v_k, v_l)$.

Canonical Parameterization (4.4.2.1). Markov networks are generally overparameterized[172]. The **canonical parameterization**, which requires that P be positive, avoids this. First, some notation and requirements:

- P must be positive.
- Let $\xi^* = (x_1^*, \ldots, x_n^*)$ denote some fixed assignment to the network variables $\mathcal{X}$.
- Define $\boldsymbol{x}_{\boldsymbol{Z}} \triangleq \boldsymbol{x}\langle \boldsymbol{Z} \rangle$ as the assignment of variables in some subset $\boldsymbol{Z}$.[173]
- Define $\xi^*_{-\boldsymbol{Z}} \triangleq \xi^* \langle \mathcal{X} - \boldsymbol{Z} \rangle$ be our fixed assignment for the variables outside $\boldsymbol{Z}$.
- Let $\ell(\xi)$ denote $\ln P(\xi)$.

[172] Meaning: for any P_Φ, there are often infinitely many ways to choose its set of parameter values for a given $\mathcal{H}$.

[173] And $\boldsymbol{x}$ is some assignment to some subset of $\mathcal{X}$ that also contains $\boldsymbol{Z}$.

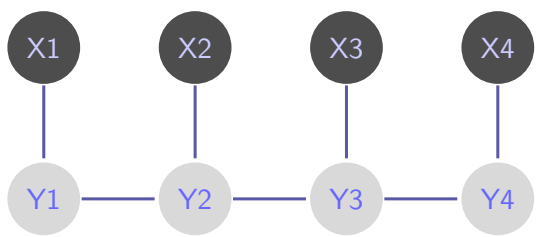
The **canonical energy function** for a clique $\boldsymbol{D}$ is defined below, as well as the associated total $P(\xi)$ over a full network assignment:

The sum is over all subsets of $\boldsymbol{D}$, including $\boldsymbol{D}$ itself and $\varnothing$.

$$\epsilon^*_{\boldsymbol{D}}(\boldsymbol{d}) = \sum_{\boldsymbol{Z} \subseteq \boldsymbol{D}} \ell(\boldsymbol{d}_{\boldsymbol{Z}}, \xi^*_{-\boldsymbol{Z}}) \cdot (-1)^{|\boldsymbol{D}-\boldsymbol{Z}|} \tag{762}$$

$$P(\xi) = \exp\left[\sum_i \epsilon^*_{\boldsymbol{D}_i}(\xi\langle \boldsymbol{D}_i \rangle)\right] \tag{763}$$

Conditional Random Fields. So far, we've only described Markov network representation as encoding a joint distribution over $\mathcal{X}$. The same undirected graph representation and parameterization can also be used to encode a *conditional distribution* $\Pr[\boldsymbol{Y} \mid \boldsymbol{X}]$, where $\boldsymbol{Y}$ is a set of *target variables* and $\boldsymbol{X}$ is a (disjoint) set of *observed variables*.



Note how there are no connection between any of the Xs

Formally, a CRF is an undirected graph $\mathcal{H}$ whose nodes correspond to $\boldsymbol{Y} \cup \boldsymbol{X}$. Since we want to avoid representing[174] a probabilistic model over $\boldsymbol{X}$, we disallow potentials that involve only variables in $\boldsymbol{X}$; our set of factors is $\phi_1(\boldsymbol{D}_1), \ldots, \phi_m(\boldsymbol{D}_m)$, such that each $\boldsymbol{D}_i \nsubseteq \boldsymbol{X}$. The network encodes a conditional distribution as follows:

$$P(\boldsymbol{Y} \mid \boldsymbol{X}) = \frac{1}{Z(\boldsymbol{X})} \widetilde{P}(\boldsymbol{Y}, \boldsymbol{X}) \tag{764}$$

$$\widetilde{P}(\boldsymbol{Y}, \boldsymbol{X}) = \prod_{i=1}^{m} \phi_i(\boldsymbol{D}_i) \tag{765}$$

$$Z(\boldsymbol{X}) = \sum_{\boldsymbol{Y}} \widetilde{P}(\boldsymbol{Y}, \boldsymbol{X}) \tag{766}$$

where now the partition function is a function of the assignment $\boldsymbol{x}$ to $\boldsymbol{X}$.

[174] Also note how we never have to deal with a summation over all possible $\boldsymbol{X}$, due to restricting ourselves to $Z(\boldsymbol{X})$.

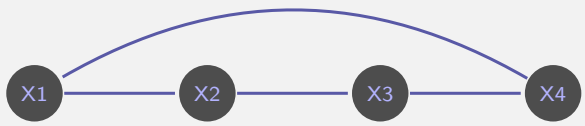
Rapid Summary.

- **Gibbs distribution**: any probability distribution that can be written as a product of factors divided by some partition function Z.
- **Factorizes**: A Gibbs distribution factorizes over $\mathcal{H}$ if each factor [in its product of factors] is a clique.

Exercises

Exercise 4.1

Let $\mathcal{H}$ be the graph of binary variables below. Show that P does not factorize over $\mathcal{H}$ (Hint: proof by contradiction).



- Example 4.4 recap: P satisfies the global independencies w.r.t $\mathcal{H}$. They showed this by manually checking the two global indeps of $\mathcal{H}$, $(X_1 \perp X_3 \mid X_2, X_4)$ and $(X_2 \perp X_4 \mid X_1, X_3)$, against the tabulated list of possible assignments for P (given in example). Nothing fancy.
- P factorizes over $\mathcal{H}$ if it can be written as a product of clique potentials.
- My proof:
  1. Assume that P *does* factorize over $\mathcal{H}$.
  2. Then P can be written as

$$P(X_1, X_2, X_3, X_4) = \frac{1}{Z}\phi_1(X_1, X_2)\phi_2(X_2, X_3)\phi_3(X_3, X_4)\phi_4(X_4, X_1) \tag{767}$$

     Let the above assertion be denoted as C, and the statement that P factorizes according to $\mathcal{H}$ be denoted simply as $P_{\mathcal{H}}$. Since $P_{\mathcal{H}} \iff C$, if we can prove that C does not hold, then we've found our contradiction, meaning $P_{\mathcal{H}}$ also must not hold.
  3. I know that the proof must take advantage of the fact that we know P is zero for certain assignments to $\mathcal{X}$. For example $P(0100) = 0$. Furthermore, by looking at the assignments where P is *not* zero, I can see that all possible combinations of (X_1, X_2) are present, which means ϕ_1 never evaluates to zero.
  4. From the example, we know that $P(1100) = 1/8 \neq 0$. However, since

$$0 = \frac{P(0100)}{P(1100)} = \frac{\phi_1(0,1)\phi_4(0,0)}{\phi_1(1,1)\phi_4(0,1)} \tag{768}$$

$$= \frac{\phi_4(0,0)}{\phi_4(0,1)} \qquad (\phi_1 > 0) \tag{769}$$

     and we also know that both the numerator and denominator of eq. 769 are positive, and thus we have a contradiction.

Exercise 4.4

Prove theorem 4.7 for the case where $\mathcal{H}$ consists of a single clique. Theorem 4.7 is equation 763 in my notes. For a single clique $\boldsymbol{D}$, the question reduces to: Show that, for any assignment $\boldsymbol{d}$ to $\boldsymbol{D}$:

$$P(\boldsymbol{d}) = \frac{\exp\Big(-\epsilon(\boldsymbol{d})\Big)}{\sum_{\boldsymbol{d}'} \exp\Big(-\epsilon(\boldsymbol{d}')\Big)} \tag{770}$$

$$= \exp\Big(\epsilon^*_{\boldsymbol{D}}(\boldsymbol{d})\Big) \tag{771}$$

Consider the case where $|\boldsymbol{D}| = 1$, i.e. $\boldsymbol{d} = d$ is a single variable. Then

$$\epsilon^*_{\boldsymbol{D}}(\boldsymbol{d}) = (-1)^{|\boldsymbol{D}|}\ell(\xi^*_{\boldsymbol{D}}) + (-1)^{|\boldsymbol{D}-\boldsymbol{D}|}\ell(\boldsymbol{d}) \tag{772}$$

$$= -\ell(\xi^*_{\boldsymbol{D}}) + \ell(\boldsymbol{d}) \tag{773}$$

$$= -\ln P(d^*) + \ln P(d) \tag{774}$$

and therefore

$$\exp\Big(\epsilon^*_{\boldsymbol{D}}(\boldsymbol{d})\Big) = P(d)/P(d^*) \tag{775}$$

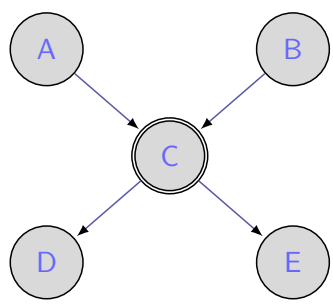
which is clearly incorrect (???) **TODO**: figure out what's going on here. Either the book has a type in its for theorem 4.7, or I'm absolutely insane.

Koller and Friedman (2009). Local Probabilistic Models.
Probabilistic Graphical Models: Principles and Techniques.

Deterministic CPDs. When X is a deterministic fu nction of its parents Pa_X:

$$P(x \mid Pa_X) = \begin{cases} 1 & x = f(Pa_X) \\ 0 & \text{otherwise.} \end{cases} \tag{776}$$

Consider the example below, where the double-line notation on C means that C is a deterministic function of A and B. What new conditional dependencies do we have?

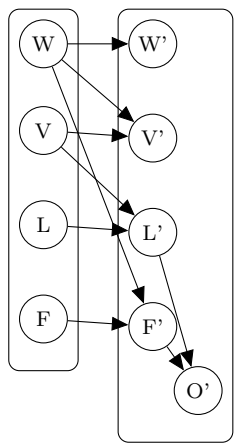


Answer: $(D \perp E \mid A, B)$, which would not be true by d-separation alone. It only holds because C is a deterministic function of A and B. z

Template-Based Representations (Ch. 6)

Koller and Friedman (2009). Template-Based Representations.
Probabilistic Graphical Models: Principles and Techniques.

In what follows, we will build on an example where a vehicle tries to track its true location (L) using various sensor readings: velocity (V), weather (W), failure of sensor (F), observed location (O) of the noisy sensor.



Temporal Models. We discretize time into slices of interval Δ, and denote the ground random variables at time $t \cdot \Delta$ by $\mathcal{X}^{(t)}$. We can simplify our formulation considerably by assuming a **Markovian system**: a dynamic system over template variables $\mathcal{X}$ that satisfies the Markov assumption:

$$(\mathcal{X}^{(t+1)} \perp \mathcal{X}^{(0:(t-1))} \mid \mathcal{X}^{(t)}) \tag{777}$$

which allows us to define a more compact representation of the joint distribution from time 0 to T:

$$P(\mathcal{X}^{(0:T)}) = P(\mathcal{X}^{(0)}) \prod_{t=0}^{T-1} P(\mathcal{X}^{(t+1)} \mid \mathcal{X}^{(t)}) \tag{778}$$

One last simplifying assumption, to avoid having unique transition probabilities for each time t, is to assume a **stationary**[175] Markovian dynamic system, defined s.t. $P(\mathcal{X}^{(t+1)} \mid \mathcal{X}^{(t)})$ is the same for all t.

[175] Also called time invariant or homogeneous.

Dynamic Bayesian Networks (6.2.2). Above, I've drawn the **2-time-slice Bayesian network** (2-TBN) for our location example. A 2-TBN is a conditional BN over $\mathcal{X}'$ given $\mathcal{X}_I$, where $\mathcal{X}_I \subseteq \mathcal{X}$ is a set of **interface variables**[176]. For each template variable X_i, the CPD $P(X_i' \mid Pa_{X_i'})$ is a **template factor**. We can use the notion of the 2-TBN to define the more general **dynamic Bayesian network**:

> *A **dynamic Bayesian network** (DBN) is a pair $\langle \beta_0, \beta_\rightarrow \rangle$, where β_0 is a Bayesian network over $\mathcal{X}^{(0)}$, representing the initial distribution over states, and $\beta_\rightarrow$ is a 2-TBN for the process. For any $T \geq 0$, the unrolled Bayesian network is defined such that*
>
> - *$p(X_i^{(0)} \mid Pa_{X_i^{(0)}})$ is the same as the CPD for the corresponding X_i in β_0.*
> - *$p(X_i^{(t)} \mid Pa_{X_i^{(t)}})$ (for $t > 0$) is the same as the CPD for the corresponding X_i' in $\beta_\rightarrow$.*

State-Observation Models (6.2.3). Temporal models that, in addition to the Markov assumption (eq. 777), model the observation variables at time t as conditionally independent of the entire state sequence given the variables at time t:

$$\left(\boldsymbol{O}^{(t)} \perp \boldsymbol{X}^{(0:(t-1))}, \boldsymbol{X}^{(t+1:\infty)} \mid \boldsymbol{X}^{(t)}\right) \tag{779}$$

So basically a 2-TBN with the constraint that observation variables are leaves and only have parents in $\boldsymbol{X}'$. We now view our probabilistic model as consisting of 2 components: the *transition model* $P(\boldsymbol{X}' \mid \boldsymbol{X})$, and the *observation model* $P(\boldsymbol{O} \mid \boldsymbol{X})$. The two main architectures for such models are as follows:

- **Hidden Markov Models**. Defined as having a single state variable S and a single observation variable O. In practice, the transition model $P(S' \mid S)$ is often assumed to be sparse (many possible transitions having zero probability). In such cases, one usually represents them visually as **probabilistic finite-state automaton**[177].
- **Linear Dynamical Systems** (LDS) represent a system of one or more real-valued variables that evolve linearly over time, with some Gaussian noise. Such systems are often called **Kalman filters**, after the algorithm used to perform tracking. They can be viewed as a DBN with continuous variables and all dependencies are linear Gaussian[178]. A LDS is traditionally represented as a state-observation model, where both state and observation are vector-valued RVs, and the transition/observation models are encoded using matrices. More formally, for $\boldsymbol{X}^{(t)} \in \mathbb{R}^n, O \in \mathbb{R}^m$:

$$P(\boldsymbol{X}^{(t)} \mid \boldsymbol{X}^{(t-1)}) = \mathcal{N}(A\boldsymbol{X}^{(t-1)}; Q) \tag{780}$$ $Q \in \mathbb{R}^{n \times n}$

$$P(O^{(t)} \mid \boldsymbol{X}^{(t)}) = \mathcal{N}(H\boldsymbol{X}^{(t)}; R) \tag{781}$$ $H \in \mathbb{R}^{m \times m}$

[176] Interface variables are those variables whose values at time t can have a direct effect on the variables at time $t+1$.

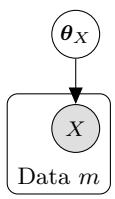
[177] FSA use the graphical notation where the nodes are the individual possible values in $Val(S)$, and the directed edges from some a *to* b have weight equal to $P(S' = b \mid S = a)$.

[178] Linear Gaussian: some var Z pointing to X denotes that $X = \Lambda Z + \text{noise}$, where noise $\sim \mathcal{N}(\mu_x, \Sigma_x)$.

Template Variables and Template Factors (6.3). It's convenient to view the world as being composed of a set of **objects**, which can be divided into a set of mutually exclusive and exhaustive *classes* $\mathcal{Q} = Q_1, \ldots, Q_k$. Template **attributes** have a tuple of *arguments*, each of which is associated with a particular class of objects, which defines the set of objects that can be used to instantiate the argument in a given domain. Template attributes thus provide us with a "generator" for RVs in a given probability space. Formally,

> *An **attribute** A is a function $A(U_1, \ldots, U_k)$, whose range is some set $Val(A)$, and where each argument U_i is a typed **logical variable** associated with a particular class $Q[U_i]$. The tuple $U_1, ldots, U_k$ is called the **argument signature** of the attribute A, and denoted $\alpha(A)$.*

Plate Models (6.4.1). The simplest example of a plate model is shown below. It describes multiple RVs generated from the same distribution $\mathcal{D}$.



This could be a plate model for a set of coin tosses sampled from a single coin. We have a set of m random variables $X(d)$, where $d \in \mathcal{D}$. Each $X(d)$ is the random variable for the dth coin toss. We also explicitly model that the single coin for which the tosses are used is sampled from a distribution θ_X, which takes on values $[0, 1]$ and denotes the bias of the coin.

Gaussian Network Models (Ch. 7)

Koller and Friedman (2009). Gaussian Network Models.
Probabilistic Graphical Models: Principles and Techniques.

Multivariate Gaussians. Here I'll give two forms of the familiar density function, followed by some comments and terminology.

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}\sqrt{\det \Sigma}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \tag{782}$$

$$p(\boldsymbol{x}) \propto \exp\left\{-\frac{1}{2}\boldsymbol{x}^T J \boldsymbol{x} + (J\boldsymbol{\mu})^T \boldsymbol{x}\right\} \quad \text{where } J \triangleq \Sigma^{-1} \tag{783}$$

- The **standard Gaussian** is defined as $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$.
- Σ must be *positive definite*[179]: $\forall \boldsymbol{x} \neq 0,\ \boldsymbol{x}^T \Sigma \boldsymbol{x} > 0$. Recall that $\Sigma_{i,j} = \mathrm{Cov}\left[x_i, x_j\right] = \mathbb{E}\left[x_i x_j\right] - \mu_i \mu_j$.

The two operations we usually want to perform on a Gaussian are (1) computing marginals, and (2) conditioning the distribution on an assignment of some subset of the variables. For (1), its easier to use the standard form of $p(\boldsymbol{x})$, whereas for (2) it is easier to use the information form (the one using J).

Multivariate Gaussians are also special because we can easily determine whether two x_i and x_j are independent: $x_i \perp x_j$ IFF $\Sigma_{i,j} = 0$[180]. For conditional independencies, the information matrix J is easier to work with: $(x_i \perp x_j \mid \{x\}_{k \notin \{i,j\}})$ IFF $J_{i,j} = 0$. This condition is also how we defined pairwise independencies in a Markov network, which leads to the awesome realization:

> *We can view the information matrix J as directly defining a minimal I-map Markov network for [multivariate Gaussian] p, whereby any entry $J_{i,j} \neq 0$ corresponds to an edge x_i—x_j in the network.*

[179] Most definitions of p.d. also require that the matrix be symmetric. I also think it helps to view positive definite from the operator perspective: *A linear operator T is positive definite if T is self-adjoint and $\langle T(x), x\rangle > 0$. In other words, "positive definite" means that the result of applying the matrix/operator to any nonzero $\boldsymbol{x}$ will always have a positive component along the original direction $\hat{\boldsymbol{x}}$.*

[180] This is not true in general – just for multivariate Gaussians!

Variable Elimination (Ch. 9)

Koller and Friedman (2009). Variable Elimination.
Probabilistic Graphical Models: Principles and Techniques.

Analysis of Exact Inference. The focus of this chapter is the conditional probability query,

$$\Pr\left[\boldsymbol{Y} \mid \boldsymbol{E} = \boldsymbol{e}\right] = \frac{\Pr\left[\boldsymbol{Y}, \boldsymbol{e}\right]}{\Pr\left[\boldsymbol{e}\right]} \tag{9.1}$$

Ideally, we want to obtain all instantiations $\Pr\left[\boldsymbol{y} \mid \boldsymbol{e}\right]$ of equation 9.1. Let $\boldsymbol{W} = \mathcal{X} - \boldsymbol{Y} - \boldsymbol{E}$ be the RVs that are neither query nor evidence. Then

$$\Pr\left[\boldsymbol{y} \mid \boldsymbol{e}\right] = \frac{\sum_{\boldsymbol{w}} \Pr\left[\boldsymbol{y}, \boldsymbol{e}, \boldsymbol{w}\right]}{\sum_{\boldsymbol{y}} \Pr\left[\boldsymbol{y}, \boldsymbol{e}\right]} \tag{9.3}$$

and note that, by computing all instantiations for the numerator first, we can reuse them to obtain the denominator. We can formulate the inference problem as a decision problem, which we will call $BNPrDP$, defined as follows:

Given: Bayesian network $\mathcal{B}$ over $\mathcal{X}$, a variable $X \in \mathcal{X}$, and a value $x \in Val(X)$.
Decide: whether $Pr_{\mathcal{B}}\left[X = x\right] > 0$. $BNPrDP$

Thm 9.1: The decision problem $BNPrDP$ is $\mathcal{NP}$-complete. **Proof**:

1. **BNPrDP is in $\mathcal{NP}$**: Guess a full assignment ξ to the network[181]. If the guess is successful, where sucess if defined as $(X = x) \in \xi$ and $P(\xi) > 0$, then we know that $P(X = x) > 0$[182]. Computing $P(\xi)$ is linear in the number of factors for a BN, since we just multiply them together.
2. **BNPrDP is $\mathcal{NP}$-hard**. We show this by proving that we can solve 3-SAT (which is $\mathcal{NP}$-hard) by transforming inputs to 3-SAT to inputs of BNPrDP in polynomial time. Given any 3-SAT formula ϕ, we can create a BN B_ϕ with some special variable X s.t. ϕ is satisfiable IFF $P_{B_\phi}(X = x^1) > 0$. You can easily build such a network by having a node Q_i for each binary RV q_i, and a node C_i for each of the clauses that's a deterministic function of its parents (up to 3 Q nodes). Then, the node X is a deterministic function of its parents, which are chains of AND gates along the C_i. Since each node has at most 3 parents, we can ensure that construction is bounded by polynomial time in the length of ϕ.

[181] Apparently the time it takes to generate a guess is irrelevant.
[182] This is true because $P(x)$ can be decomposed as $P(\xi) + \sum P(\ldots) \geq P(\xi)$.

Analysis of Approximate Inference. Consider a specific query $P(\boldsymbol{y} \mid \boldsymbol{e})$, where we focus on a particular assignment $\boldsymbol{y}$. Let ρ denote some approximate answer, whose accuracy we wish to evaluate relative to the correct probability. We can use the **relative error** to estimate the quality of the approximation: *An estimate ρ has relative error ϵ if:*

$$\frac{\rho}{1+\epsilon} \leq P(\boldsymbol{y} \mid \boldsymbol{e}) \leq \rho(1+\epsilon) \tag{784}$$

Unfortunately, the task of finding some approximation ρ with relative error ϵ *is also $\mathcal{NP}$-hard*. Furthermore, even if we relax this metric by using absolute error instead, we end up finding that **in the case where we have evidence, approximate inference is no easier than exact inference, in the worst case**.

Variable Elimination. Basically, a dynamic programming approach for performing exact inference. Consider the simple BN $X_1 \to \cdots \to X_n$, where each variable can take on k possible values. The dynamic programming approach for computing $P(X_n)$ involves computing

$$P(X_{i+1}) = \sum_{x_i} P(X_{i+1} \mid x_i) P(x_i) \tag{785}$$

$n-1$ times, starting with $i=1$, all the way up to $i=n-1$, reusing the previous computation at each step, with total cost $\mathcal{O}(nk^2)$. So for this simple network, even though the size of the joint is k^n (exponential in n), we can do inference in linear time.

First, we formalize some basic concepts before defining the algorithm.

Factor marginalization:

> *Let $\boldsymbol{X}$ be a set of variables, and $Y \notin \boldsymbol{X}$ a variable. Let $\phi(\boldsymbol{X}, Y)$ be a factor. Define the **factor marginalization** of Y in ϕ, denoted $\sum_Y \phi$, to be a factor ψ over $\boldsymbol{X}$ such that:*
>
> $$\psi(\boldsymbol{X}) = \sum_Y \phi(\boldsymbol{X}, Y)$$

The key observation that's easy to miss is that *we're only summing entries in the table where the values of $\boldsymbol{X}$ match up*. One useful rule for exchanging factor product and summation: If $X \notin Scope[\phi_1]$, then

$$\sum_X (\phi_1 \cdot \phi_2) = \phi_1 \cdot \sum_X \phi_2 \tag{9.6}$$

So, when computing some marginal probability, the main idea is to group factors together and compute expressions of the form

$$\sum_{\boldsymbol{Z}} \prod_{\phi \in \Phi} \phi \tag{786}$$

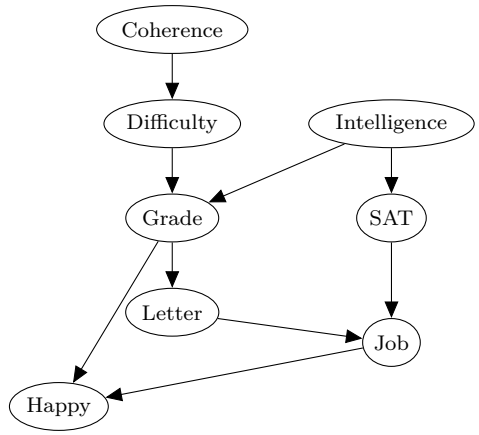
where Φ is the set of all factors ϕ for which $\boldsymbol{Z} \in Scope[\phi]$. This is commonly called the **sum-product** inference task. The full algorithm for sum-product variable elimination, which is an instantiation of the sum-product inference task, is illustrated below.

Procedure Sum-Product-VE (
  Φ, // Set of factors
  $\boldsymbol{Z}$, // Set of variables to be eliminated
  $\prec$ // Ordering on $\boldsymbol{Z}$
)
  Let $Z_1, \ldots, Z_k$ be an ordering of $\boldsymbol{Z}$ such that
    $Z_i \prec Z_j$ if and only if $i < j$
  for $i = 1, \ldots, k$
    $\Phi \leftarrow$ Sum-Product-Eliminate-Var(Φ, Z_i)
  $\phi^* \leftarrow \prod_{\phi \in \Phi} \phi$
  return ϕ^*

Procedure Sum-Product-Eliminate-Var (
  Φ, // Set of factors
  Z // Variable to be eliminated
)
  $\Phi' \leftarrow \{\phi \in \Phi : Z \in Scope[\phi]\}$
  $\Phi'' \leftarrow \Phi - \Phi'$
  $\psi \leftarrow \prod_{\phi \in \Phi'} \phi$
  $\tau \leftarrow \sum_{Z} \psi$
  return $\Phi'' \cup \{\tau\}$

This is what we use to compute the marginal probability $P(\boldsymbol{X})$ where $\boldsymbol{X} = \mathcal{X} - \boldsymbol{Z}$. To compute conditional queries of the form $P(\boldsymbol{Y} \mid \boldsymbol{E} = \boldsymbol{e})$, simply replace all factors whose scope overlaps with $\boldsymbol{E}$ with their reduced factor (see chapter 4 notes for definition) to get the unnormalized $\phi^*(\boldsymbol{Y})$ (the numerator of $P(\boldsymbol{Y} \mid \boldsymbol{e})$). Then divide by $\sum_{\boldsymbol{y}} \phi^*$ to obtain the final result.

Example. We will work through computing $P(Job)$ for the BN below.



Due to Happy being a child of Job, $P(J)$ actually requires using all factors in the graph. Below shows how VE with elimination ordering C, D, I, H, G, S, L progressively simplifies the equation for computing $P(J)$.

$$
\begin{aligned}
P(J) &= \sum P(J \mid s, \ell) P(s \mid i) P(\ell \mid g) P(g \mid d, i) P(d \mid c) P(h \mid J, g) P(c) P(i) && (787)\\
&= \sum_{c,d,i,h,g,s,\ell} (P(c)P(d \mid c)) \cdot P(g \mid d, i) P(i) P(s \mid i) P(h \mid J, g) P(\ell \mid g) P(J \mid s, \ell) && (788)\\
&= \sum_{d,i,h,g,s,\ell} (\tau_1(d) P(g \mid d, i)) \cdot P(i) P(s \mid i) P(h \mid J, g) P(\ell \mid g) P(J \mid s, \ell) && (789)\\
&= \sum_{i,h,g,s,\ell} (\tau_2(g, i) P(i) P(s \mid i)) \cdot P(h \mid J, g) P(\ell \mid g) P(J \mid s, \ell) && (790)\\
&= \sum_{h,g,s,\ell} (P(h \mid J, g)) \cdot \tau_3(g, s) P(\ell \mid g) P(J \mid s, \ell) && (791)\\
&= \sum_{g,s,\ell} (\tau_4(g, J) \tau_3(g, s) P(\ell \mid g)) \cdot P(J \mid s, \ell) && (792)\\
&= \sum_{s,\ell} \tau_5(J, \ell, s) \cdot P(J \mid s, \ell) && (793)\\
&= \sum_{\ell} \tau_6(J, \ell) && (794)
\end{aligned}
$$

where red indicates the focus of the given step in the VE algorithm.

Probabilistic Graphical Models **May 06, 2018**

Clique Trees (Ch. 10)

Table of Contents Local *Written by Brandon McKinzie*

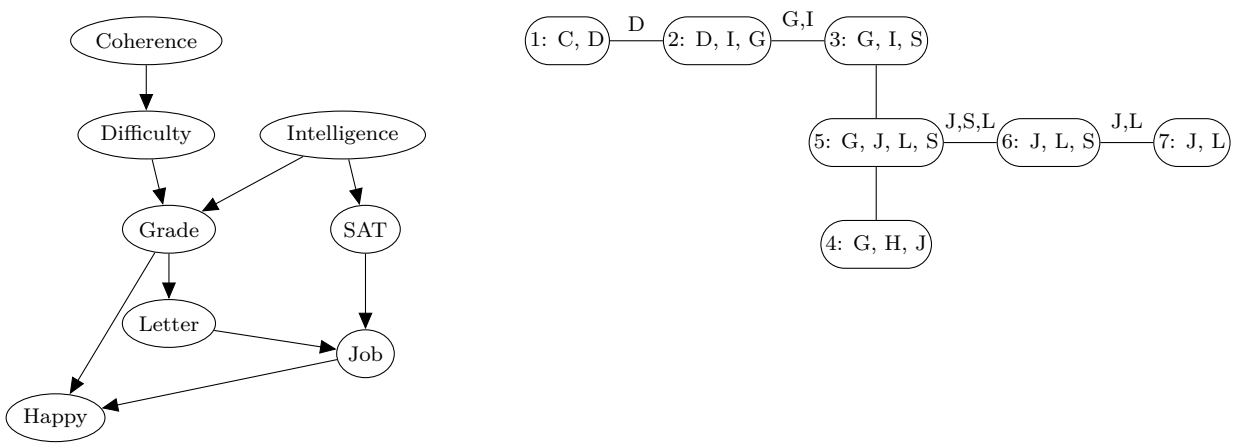
Koller and Friedman (2009). Clique Trees.
Probabilistic Graphical Models: Principles and Techniques.

Cluster Graphs. A graphical flowchart of the factor-manipulation process that will be relevant when we discuss message passing. Each node is a *cluster*, which is associated with a subset of variables. Formally,

> *A **cluster graph** $\mathcal{U}$ for a set of factors Φ over $\mathcal{X}$ is an undirected graph. Each node i is associated with a subset $\boldsymbol{C}_i \subseteq \mathcal{X}$. Each factor $\phi \in \Phi$ must be associated with a cluster $\boldsymbol{C}_i$, denoted $\alpha(\phi)$, such that $Scope[\phi] \subseteq \boldsymbol{C}_i$ (**family-preserving**). Each edge between a pair of clusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ is associated with a **sepset** $\boldsymbol{S}_{i,j} \subseteq \boldsymbol{C}_i \cap \boldsymbol{C}_j$.*

Recall that each step of variable elimination involves creating a factor ψ_i by multiplying a group of factors[183]. Then, denoting the variable we are eliminating at this step as Z, we obtain another factor τ_i that's the factor marginalization of Z in ψ_i (denoted $\sum_Z \psi_i$). An execution of variable elimination defines a cluster graph: we have a cluster for each of the ψ_i, defined as $\boldsymbol{C}_i = Scope[\psi_i]$. We draw an edge between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ if the **message** τ_i is used in the computation of τ_j.

Consider when we applied variable elimination to the student graph network below, to compute $P(J)$. Elimination ordering C, D, I, H, G, S, L.



[183] All whose scope contains the variable we are currently trying to eliminate.

Clique Trees. Since VE uses each intermediate τ_i at most once, the cluster graph induced by an execution VE is necessarily a *tree*, and it also defines a directionality: the direction of the message passing (left-to-right in the above illustration). All the messages flow toward a single cluster where the final result is computed – the **root** of the tree; we say the messages "flow up" to the root. Furthermore, for cluster trees induced by VE, the scope of each message (edge) τ_i is *exactly* $\boldsymbol{C}_i \cap \boldsymbol{C}_j$, not just a subset[184].

> *Let Φ be a set of factors over $\mathcal{X}$. A cluster tree over Φ that satisfies the running intersection property is called a* ***clique tree****. For clique trees, the clusters are also called* ***cliques****.*

Message Passing: Sum Product. Previously we saw how an execution of VE can be illustrated with a clique tree. We now go the other direction – given a clique tree, we show how it can be used for variable elimination. Given a clique tree representation of some BN, we can use it to guide us along an execution of VE to compute any marginal we'd like. First, before any run, we generate the set of **initial potentials** ψ_i associated with each clique $\boldsymbol{C}_i$ in the tree, defined as just the multiplication of the initial factors associated with the clique. We define the root of the tree as any clique containing the variable whose marginal we want to compute (we pick arbitrarily). Starting from the leaves and moving toward the root, we pass messages along from clique to clique. A clique is *ready* to send a message when it has received a message from all of its downstream neighbors. The message from $\boldsymbol{C}_i$ to [a neighbor] $\boldsymbol{C}_j$ is computed using the **sum-product message passing** computation:

$$\delta_{i \to j} = \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in (Nb_i - \{j\})} \delta_{k \to i} \tag{10.2}$$

where the summation is simply over the variables in $\boldsymbol{C}_i$ that aren't passed along to $\boldsymbol{C}_j$, and the product is over all messages that $\boldsymbol{C}_i$ received. Stated even simpler, we multiply all the incoming messages by our initial potential, then sum out all variables except those in $\boldsymbol{S}_{i,j}$. When the root clique has received all messages, it multiplies them with its own initial potential, resulting in a factor called the **beliefs**, $\beta_r(\boldsymbol{C}_r)$. It represents

$$\widetilde{P}_\Phi(\boldsymbol{C}_r) = \sum_{\mathcal{X} - \boldsymbol{C}_r} \prod_{\phi} \phi \tag{795}$$

where, to be clear, the product is over all ϕ in the graph.

[184] This follows from the **running intersection property**, which is satisfied by any cluster tree that's defined by variable elimination. It's defined as, if any variable X is in both cluster $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$, then X is also in every cluster in the (unique) path in the tree between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$.

Below is a more compact summary of all of this, showing the procedure for computing *all* final factors (belief) β_i for some marginal probability query on the variables in $\boldsymbol{C}_r$ *asynchronously.*

Algorithm 10.2: Sum-Product Belief Propagation

1. For each clique $\boldsymbol{C}_i$, compute its initial potential:

$$\psi_i(\boldsymbol{C}_i) \leftarrow \prod_{\phi_j : \alpha(\phi_j) = i} [\phi_j]$$

2. While $\exists i, j$ such that i is ready to transmit to j, compute:

$$\delta_{i \to j} \leftarrow \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \psi_i \cdot \prod_{k \in (Nb_i - \{j\})} \delta_{k \to i}$$

3. Then, compute each belief factors β_i by multiplying the initial potential ψ_i by the incoming messages to $\boldsymbol{C}_i$:

$$\beta_i \leftarrow \psi_i \cdot \prod_{k \in Nb_{C_i}} \delta_{k \to i}$$

4. Return the set of beliefs $\{\beta_i\}$, where

$$\beta_i = \sum_{\mathcal{X} - \boldsymbol{C}_i} \widetilde{P}_\Phi(\mathcal{X}) = \widetilde{P}_\Phi(\boldsymbol{C}_i) \tag{796}$$

The SP Belief Propagation algorithm above is also called **clique tree calibration**. A clique tree $\mathcal{T}$ is *calibrated* if all pairs of adjacent cliques are calibrated. A calibrated clique tree satisfies the following property for what we'll now call the **clique beliefs**, β_i, and the **sepset beliefs**, $\mu_{i,j}$ over $\boldsymbol{S}_{i,j}$:

$$\mu_{i,j}(\boldsymbol{S}_{i,j}) \triangleq \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i = \sum_{\boldsymbol{C}_j - \boldsymbol{S}_{i,j}} \beta_j \tag{797}$$

$\mu_{i,j} = \widetilde{P}_\Phi(\boldsymbol{S}_{i,j})$

> *The main advantage of the clique tree algorithm is that it computes the posterior probability of all variables in a graphical model using only twice the computation*[185] *of the upward pass in the same tree.*

We can also show that $\mu_{i,j} = \delta_{j \to i} \delta_{i \to j}$, which then allows us to derive:

$$\widetilde{P}_\Phi(\mathcal{X}) = \frac{\prod_{i \in \mathcal{V}_\mathcal{T}} \beta_i}{\prod_{ij \in \mathcal{E}_\mathcal{T}} \mu_{i,j}} \tag{10.10}$$

In other words, the clique and sepset beliefs provide a **reparameterization** of the unnormalized measure, a property called the **clique tree invariant**.

[185] The algorithm is equivalent to doing one upward pass, one downward pass.

Message Passing: Belief Update. We know discuss an alternative message passing approach that is mathematically equivalent but intuitively different than the sum-product approach. First, we introduce some new definitions.

Factor Division:

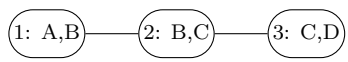
Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be disjoint sets of variables, and let $\phi_1(\boldsymbol{X}, \boldsymbol{Y})$ and $\phi_2(\boldsymbol{Y})$ be two factors. We define the division of ϕ_1 and ϕ_2 as a factor ψ with scope $\boldsymbol{X}, \boldsymbol{Y}$ as follows: Define $0/0 = 0$

$$\psi(\boldsymbol{X}, \boldsymbol{Y}) \triangleq \frac{\phi_1(\boldsymbol{X}, \boldsymbol{Y})}{\phi_2(\boldsymbol{Y})}$$

Looking back at equation 10.2, we can now see that another way to write $\delta_{i \to j}$ is

$$\delta_{i \to j} = \frac{\sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} \beta_i}{\delta_{j \to i}} \tag{10.13}$$

Now, consider the clique tree below for the simple Markov network A-B-C-D:



If we assigned $\boldsymbol{C}_2$ as the root, then our previous approach would compute $\delta_{2 \to 1}$ as $\sum_C \psi_2 \cdot \delta_{3 \to 2}$. Alternatively, we can use equation 10.13 to realize this is equivalent to dividing β_2 by $\delta_{1 \to 2}$ and marginalizing out C. This observation motivates the algorithm below, which allows us to execute message passing in terms of the clique and sepset beliefs, without having to remember the initial potentials ψ_i or explicitly compute the messages $\delta_{i \to j}$.

Algorithm 10.3: Belief-Update Message Passing

1. For each clique $\boldsymbol{C}_i$, set its initial belief β_i to its initial potential ψ_i. For each edge in $\mathcal{E}_{\mathcal{T}}$, set $\mu_{i,j} = \mathbf{1}$.
2. Wile there exists an uninformed[186] clique in $\mathcal{T}$, select any edge in $\mathcal{E}_{\mathcal{T}}$, and compute

$$\sigma_{i \to j} \leftarrow \sum_{\boldsymbol{C}_i - \boldsymbol{S}_{i,j}} [\beta_i] \tag{798}$$

$$\beta_j \leftarrow \beta_j \cdot \frac{\sigma_{i \to j}}{\mu_{i,j}} \tag{799}$$

$$\mu_{i,j} \leftarrow \sigma_{i \to j} \tag{800}$$

3. Return the resulting set of informed beliefs $\{\beta_i\}$.

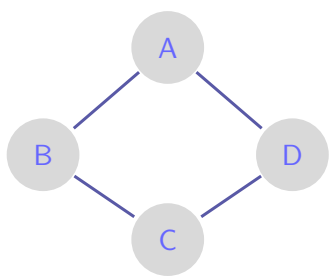
At convergence, $\sigma_{i \to j} = \mu_{i,j} = \sigma_{j \to i}$.

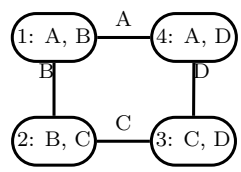
[186] A clique is informed once it has received informed messages from all of its neighbors. An informed message is one that has been sent by taking into account information from all of the sending cliques' neighbors (aside from the receiving clique of that message, of course).

Inference as Optimization (Ch. 11)

Koller and Friedman (2009). Inference as Optimization.
Probabilistic Graphical Models: Principles and Techniques.

Propagation-Based Approximation. We can use a general-purpose cluster graph rather than the more restrictive clique tree (needed to guarantee exact inference) for approximate inference methods. Consider the simple Markov network below on the left.





The clique tree for this network, which can be used for exact inference, has two cliques ABD and BCD and messages are passed between them consisting of $\tau(B, D)$. Suppose that, instead, we set up 4 clusters corresponding to each of the initial potentials, shown as the cluster graph above on the right. We can still apply belief propagation here, but due to it now having loops (as opposed to before when we only had trees), *the process may not converge*.

Parameter Estimation (Ch. 17)

Koller and Friedman (2009). Parameter Estimation.
Probabilistic Graphical Models: Principles and Techniques.

Maximum Likelihood Estimation (17.1). In this chapter, assume the network structure is fixed and that our data set $\mathcal{D}$ consists of fully observed instances of the network variables: $\mathcal{D} = \{\xi[1], \ldots, \xi[M]\}$. We begin with the simplest learning problem: parameter learning for a single variable. We want to estimate the probability, denoted via the *parameter* θ, with which the flip of a thumbtack will land heads or tails. Define the **likelihood function** $L(\theta : \boldsymbol{x})$ as the probability of observing some sequence of outcomes $\boldsymbol{x}$ under the parameter θ. In other words, it is simply $P(\boldsymbol{x} : \theta)$, but interpreted as a *function of* θ. For our simple case, where $\mathcal{D}$ consists of M thumbtack flip outcomes,

$$L(\theta : \mathcal{D}) = \theta^{M[1]}(1-\theta)^{M[0]} \tag{801}$$

where $M[1]$ denotes the number of outcomes in $\mathcal{D}$ that were heads. Since it's easier to maximize a logarithm, and since it yields the same optimal $\hat{\theta}$, optimize the **log-likelihood** to obtain:

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta : \mathcal{D}) = \arg\max_{\theta} \left[M[1]\log\theta + M[0]\log(1-\theta)\right] \tag{802}$$

$$= \frac{M[1]}{M[1]+M[0]} \tag{803}$$

Note that MLE has the *disadvantage* that it can't communicate confidence of an estimate[187].

We now provide the more general formal definitions for MLE.

- We are given a *training set* $\mathcal{D}$ containing M (IID) instances of a set of random variables $\mathcal{X}$, where the samples of $\mathcal{X}$ are drawn from some unknown distribution $P^*(\mathcal{X})$.
- We are given a **parametric model**, defined by a function $P(\xi; \boldsymbol{\theta})$, where ξ is an instance of $\mathcal{X}$, and we want to estimate its parameters $\boldsymbol{\theta}$[188]. The model also defines the space of legal parameter values Θ, the **parameter space**.
- We then define the **likelihood function** $L(\boldsymbol{\theta} : \mathcal{D}) = \prod_m P(\xi[m] : \boldsymbol{\theta})$.

[187] We get the same result (0.3) if we get 3 heads out of 10 flips, as we do for getting 300 heads out of 1000 flips; yet, the latter experiment should include a higher degree of confidence

[188] We also have the constraint that $P(\xi; \boldsymbol{\theta})$ must be a valid distribution (nonnegative and sums to 1 over all possible ξ)

We can often simplify the likelihood function to simpler terms, like our $M[0]$ and $M[1]$ values in the thumbtack example. These are called the **sufficient statistics**, defined as functions of the data that summarize the relevant information for computing the likelihood. Formally,

*A function $\tau(\xi) : \xi \to \mathbb{R}^\ell$ (for some ℓ) is a **sufficient statistic** if for any two data sets $\mathcal{D}$ and $\mathcal{D}'$, we have that*

$$\left[\sum_{\xi[m]\in\mathcal{D}} \tau(\xi[m]) = \sum_{\xi'[m]\in\mathcal{D}'} \tau(\xi'[m])\right] \implies \left[L(\boldsymbol{\theta} : \mathcal{D}) = L(\boldsymbol{\theta} : \mathcal{D}')\right] \tag{804}$$

We often informally refer to the tuple $\sum_{\xi[m]\in mathcalD} \tau(\xi[m])$ as the sufficient statistics of the data set $\mathcal{D}$.

MLE for Bayesian Networks – Simple Example. We now move on to estimating parameters $\boldsymbol{\theta}$ for the simple BN $X \to Y$ for two binary RVs X and Y. Our parameters $\boldsymbol{\theta}$ are the individual probabilities of $P(X)$ and $P(Y \mid X)$ (6 total). Since BNs have the nice property that their joint probability decomposes into a product of probabilities, just like how the likelihood function is a product of probabilities, we can write the likelihood function as a product of the individual *local* probabilities:

$$L(\boldsymbol{\theta} : \mathcal{D}) = \left(\prod_m P(x[m] : \boldsymbol{\theta}_X)\right)\left(\prod_m P(y[m] \mid x[m] : \boldsymbol{\theta}_{Y|X})\right) \tag{805}$$

decomposability of the likelihood function

which can be decomposed even further by e.g. differentiating products over $x[m] : x[m] = x^0$ etc. Just as we used $M[0]$ in the thumbtack example to count the number of instances with a certain value, we can use the same idea for the general case.

Let $\boldsymbol{Z}$ be some set of RVs, and $\boldsymbol{z}$ be some instantiation to them. We define $M[\boldsymbol{z}]$ to be the number of entries in data set $\mathcal{D}$ that have $\boldsymbol{Z}[m] = \boldsymbol{z}$:

$$M[\boldsymbol{z}] = \sum_m \mathbb{1}\{\boldsymbol{Z}[m] = \boldsymbol{z}\} \tag{806}$$

Global Likelihood Decomposition. We now move to the more general case of computing the likelihood for BN with structure $\mathcal{G}$.

$$\begin{aligned}
L(\boldsymbol{\theta} : \mathcal{D}) &= \prod_m P_{\mathcal{G}}(\xi[m] : \boldsymbol{\theta}) && (807)\\
&= \prod_m \prod_i P(x_i[m] \mid Pa_{X_i}[m] : \boldsymbol{\theta}) && (808)\\
&= \prod_i \left[\prod_m P(x_i[m] \mid Pa_{X_i}[m] : \boldsymbol{\theta}_{X_i|Pa_{X_i}})\right] && (809)\\
&= \prod_i L_i(\boldsymbol{\theta}_{X_i|Pa_{X_i}} : \mathcal{D}) && (810)
\end{aligned}$$

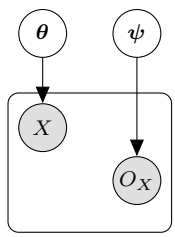
global decomposition of the likelihood

where L_i is the **local likelihood function** for X_i. Assuming these are each disjoint sets of parameters from one another, it implies that $\hat{\boldsymbol{\theta}} = \langle \hat{\boldsymbol{\theta}}_{X_1|Pa_{X_1}}, \ldots, \hat{\boldsymbol{\theta}}_{X_n|Pa_{X_n}} \rangle$

Partially Observed Data (Ch. 19)

Koller and Friedman (2009). Partially Observed Data.
Probabilistic Graphical Models: Principles and Techniques.

Likelihood of Data and Observation Models (19.1.1). Consider the simple example of flipping a thumbtack, but occasionally the thumbtack rolls off the table. We choose to ignore the tosses for which the thumbtack rolls off. Now, in addition to the random variable X giving the flip outcome, we have the *observation variable* O_X, which tells us whether we observed the value of X.



The illustration above is a plate model where we choose a thumbtack sampled with bias $\boldsymbol{\theta}$ and repeat some number of flips with that same thumbtack. We also sample the random variable O_X that has probability of observation sampled from ψ and fixed for all the experiments we do. This leads to the following definition for the observability model.

> *Let $\boldsymbol{X} = \{X_1, \dots X_n\}$ be some set of RVs, and let $O_{\boldsymbol{X}} = \{O_{X_1}, \dots, O_{X_n}\}$ be their* ***observability variable****. The* ***observability model*** *is a join distribution*
>
> $$P_{missing}(\boldsymbol{X}, O_{\boldsymbol{X}}) = P(\boldsymbol{X}) \cdot P_{missing}(O_{\boldsymbol{X}} \mid \boldsymbol{X})$$
>
> *so that $P(\boldsymbol{X})$ is parameterized by $\boldsymbol{\theta}$ and $P_{missing}(O_{\boldsymbol{X}} \mid \boldsymbol{X})$ is parameterized by $\boldsymbol{\psi}$. We define a new set of RVs $\boldsymbol{Y} = \{Y_1, \dots Y_n\}$ where $Val(Y_i) = Val(X_i) \cup \{?\}$. The actual observation $\boldsymbol{Y}$ is a deterministic function of $\boldsymbol{X}$ and $O_{\boldsymbol{X}}$:*
>
> $$Y_i = \begin{cases} X_i & O_{X_i} = o^1 \\ ? & O_{X_i} = o^0 \end{cases} \tag{811}$$

For our simple model above, we have

$$P(Y = 1) = \theta\psi \tag{812}$$
$$P(Y = 0) = (1 - \theta)\psi \tag{813}$$
$$P(Y = ?) = (1 - \psi) \tag{814}$$
$$L(\theta, \psi; \mathcal{D}) = \theta^{M[1]}(1 - \theta)^{M[0]}\psi^{M[1]+M[0]}(1 - \psi)^{M[?]} \tag{815}$$

The main takeaway is to understand that **when we have missing data, the data-generation process involves two steps: (1) generate data by sampling from the model, then (2) determine which values we get to observe and which ones are hidden from us**.

The Likelihood Function (19.1.3). Assume we have a BN network $\mathcal{G}$ over a set of variables $\boldsymbol{X}$. In general, each instance has a different set of observed variables. Denote by $\boldsymbol{O}[m]$ and $\boldsymbol{o}[m]$ the observed vars and their values in the m'th instance, and by $\boldsymbol{H}[m]$ the missing (or hidden) vars in the m'th instance.

Information Theory, Inference, and Learning Algorithms

Contents

8.1 Introduction to Information Theory (Ch. 1) 273

8.2 Probability, Entropy, and Inference (Ch. 2) 275

8.2.1 More About Inference (Ch. 3 Summary) 276

8.3 The Source Coding Theorem (Ch. 4) 278

8.3.1 Data Compression and Typicality 280

8.3.2 Further Analysis and Q&A 282

8.4 Monte Carlo Methods (Ch. 29) 284

8.5 Variational Methods (Ch. 33) 286

Introduction to Information Theory (Ch. 1)

 Written by Brandon McKinzie

[Note: Skipping most of this chapter since it's mostly introductory material.]

Preface. For ease of reference, some common quantities we will frequently be using:

- **Binomial distribution**. Let r denote the number of successful trials out of N total trials. Let f denote the probability of success for a single trial.

$$\Pr[r \mid f, N] = \binom{N}{r} f^r (1-f)^{N-r} \qquad \mathbb{E}[r] = Nf \qquad \mathrm{Var}[r] = Nf(1-f) \tag{816}$$

- **Stirling's Approximation**.

$$x! \simeq x^x e^{-x}\sqrt{2\pi x} \quad \Leftrightarrow \quad \ln x! \simeq x \ln x - x + \frac{1}{2}\ln 2\pi x \tag{817}$$

Recall that $\log_b x = \frac{\log_a x}{\log_a b}$

$$\ln \binom{N}{r} \simeq r \ln \frac{N}{r} + (N-r)\ln\frac{N}{N-r} \tag{818}$$

- **Binary Entropy Function** and its relation with Stirling's approximation.

$$H_2(x) \triangleq x \lg \frac{1}{x} + (1-x)\lg\frac{1}{1-x} \tag{819}$$

$$\lg \binom{N}{r} \simeq N H_2(r/N) \tag{820}$$

Perfect communication over an imperfect, noisy communication channel (1.1). We want to make an encoder-decoder architecture, of the general form in the figure below, to achieve reliable communication over a noisy channel.

Information theory is concerned with the theoretical limitations and potentials of such systems. Let's explore some examples for the case of the **binary symmetric channel**[189]:

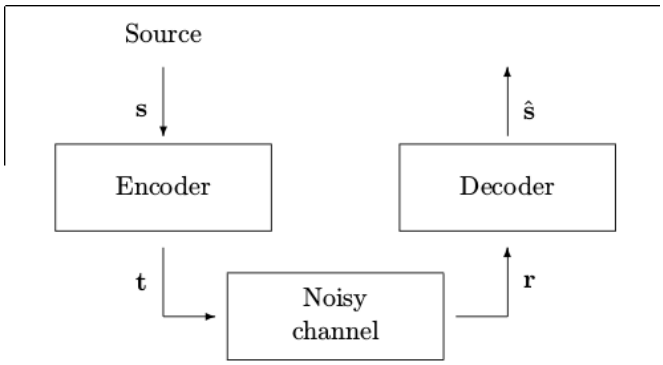
- **Repetition codes**. Let R_N denote the repetition code that repeats each bit in the message N times[190]. We model the channel as "adding[191]" a sparse noise vector $\boldsymbol{n}$ to the encoded message $\boldsymbol{t}$, so $\boldsymbol{r} := \boldsymbol{n} + \boldsymbol{t}$. What is the optimal way of decoding r?

$$\hat{s}_i \leftarrow \arg\max_{s_i} \Pr[s_i \mid \boldsymbol{r}_{i:i+n}] = \arg\max_{s_i} \Pr[\boldsymbol{r}_{i:i+N} \mid s_i]\Pr[s_i] \tag{821}$$

[189] A binary symmetric channel transmits each bit correctly with probability (1 - f) and incorrectly with probability f, where f is assumed to be small.

[190] So R_2 would encode 101 as 110011.

[191] We add in modulo 2, which is NOT the same as binary arithmetic (no carry). Addition modulo 2 is the same as doing XOR.

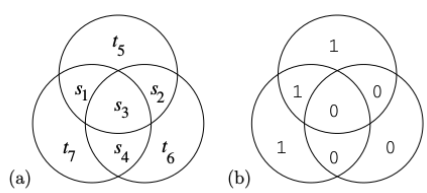


We see that we must make assumptions about the prior probability $\Pr[s_i]$. It is common to assume all possible values of s_i (0 or 1 in this case) are equally probable. It is useful to observe the **likelihood ratio**,

$$\frac{\Pr[\boldsymbol{r}_{i:i+N} \mid s_i = 1]}{\Pr[\boldsymbol{r}_{i:i+N} \mid s_i = 0]} = \prod_{n=i}^{i+N-1} \frac{\Pr[r_n \mid t_n = 1]}{\Pr[r_n \mid t_n = 0]} = \prod_{n=i}^{i+N-1} \begin{cases} \gamma & \text{if } r_n = 1 \\ \gamma^{-1} & \text{if } r_n = 0 \end{cases} \tag{822}$$

where we've defined $\gamma := (1-f)/f$, with f being the probability of a bit getting flipped by the channel. We want to assign $\hat{s}_i$ to the most likely **hypothesis** out of the possible s_i. If the likelihood ratio [for the two hypotheses] is greater than 1, we choose $\hat{s}_i = 1$, else we choose $\hat{s}_i = 0$.

- **Block Codes - the (7, 4) Hamming Code**. Although, by increasing the number of repetitions per bit N for our R_N repetition code can decrease the error-per-bit probability p_b, we incur a substantial decrease in the *rate of information transfer* – a factor of 1/N. The **(7, 4) Hamming Code** tries to improve this by encoding *blocks* of bits at a time (instead of per-bit). It is a *linear block code* – it encodes each 4-bit block into a 7-bit block, where the additional 3 bits are linear functions of the original $K = 4$ bits. The $(7, 4)$ Hamming code has each of the extra 3 bits act as parity checks. It's easiest to show this with the illustration below:



which encodes `1000` as `1000101`.

Probability, Entropy, and Inference (Ch. 2)

 Written by Brandon McKinzie

[Note: Skipping most of this chapter since it's mostly introductory material.]

Notation. Some of the notation this author seems to use a lot.

- **Ensemble** X: a triple $(x, \mathcal{A}_X, \mathcal{P}_X)$, where the outcome x is the value of a R.V. which can take on one of a set of possible values ("alphabet"), $\mathcal{A}_X = \{a_1, \ldots, a_i, \ldots, a_I\}$, having probabilities $\mathcal{P}_X = \{p_1, \ldots, p_i, \ldots p_I\}$. Note that this doesn't appear technically consistent with how the author actually *uses* the term ensemble – in practice, he actually means "X is the <u>set of all possible</u> triples $(x, \mathcal{A}_X, \mathcal{P}_X), \forall x \in \mathcal{A}_X$, or something like that. He uses it to casually refer to the space of possibilities.

Forward Probabilities and Inverse Probabilities. Both of these involve a **generative model** of the data. In a *forward* probability problem, we want find the PDF, expectation, or some other function of a quantity that depends on the data/is *produced* by the generative process. For example, we can model a series of N coin flips as "producing" the quantity n_H, denoting the number of heads. In an *inverse* probability problem, we want to compute conditional probabilities on one or more of the *unobserved variables* in the process, *given* the observed variables.

The Likelihood Principle. For a generative model on data d given parameters $\boldsymbol{\theta}$, $\Pr[d \mid \boldsymbol{\theta}]$, and having observed a particular outcome d_1, all inferences and predictions should depend only on the function $\Pr[d_1 \mid \boldsymbol{\theta}]$.

Entropy and Related Quantities.

- **Shannon information content** of an outcome x:

$$h(x) \triangleq \lg\left(\frac{1}{P(x)}\right) \tag{823}$$

- **Entropy of Ensemble** X. Defined to be the average Shannon information content of an outcome.

$0 \times \lg\left(\frac{1}{0}\right) \triangleq 0$

$$H(X) \triangleq \sum_{x \in \mathcal{A}_X} \Pr[x] \lg\left(\frac{1}{\Pr[x]}\right) \tag{824}$$

$$H(X) \leq \lg(|\mathcal{A}_X|) \quad \text{with equality iff} \quad p_i = \frac{1}{\mathcal{A}_X} \ \forall i \tag{825}$$

- **Decomposability of the Entropy**. For any probability distribution $\boldsymbol{p} = \{p_1, p_2, \ldots, p_I\}$ and m (where $1 \leq m \leq I$):

$$H(\boldsymbol{p}) = H\left(\Sigma_{1:m}, \Sigma_{m+1:I}\right) \tag{826}$$
$$+ \Sigma_{1:m} H\left(\frac{p_1}{\Sigma_{1:m}}, \ldots, \frac{p_m}{\Sigma_{1:m}}\right) \tag{827}$$
$$+ \Sigma_{m+1:I} H\left(\frac{p_{m+1}}{\Sigma_{m+1:I}}, \ldots, \frac{p_I}{\Sigma_{m+1:I}}\right) \tag{828}$$

  where I've let $\Sigma_{1:m} := \sum_{i=1}^{m} p_i$.
- **Kullback-Leibler Divergence** between two probability distributions $P(X)$ and $Q(X)$ that are defined over the same alphabet $\mathcal{A}_X$:

$$D_{KL}(P||Q) = \sum_x P(x) \lg \frac{P(x)}{Q(x)} \tag{829}$$
$$D_{KL}(P||Q) \geq 0 \qquad \textbf{[Gibb's Inequality]} \tag{830}$$

  where, in the words of the author, "**Gibb's inequality is probably the most important inequality in this book**".
- **Convex functions and Jensen's Inequality**. A function $f(x)$ is convex over the interval $[x = a, x = b]$ if every chord of the function lies above the function. That is, $\forall x_1, x_2 \in [a, b]$ and $0 \leq \lambda \leq 1$:

$$f\left(\lambda x_1 + (1-\lambda)x_2\right) \leq \lambda f(x_1) + (1-\lambda) f(x_2) \tag{831}$$
$$f\left(\mathbb{E}\left[x\right]\right) \leq \mathbb{E}\left[f(x)\right] \qquad \textbf{[Jensen's Inequality]} \tag{832}$$

  and we say f is **strictly convex** if, $\forall x_1, x_2 \in [a, b]$, we get equality for only $\lambda = 0$ and $\lambda = 1$.

More About Inference (Ch. 3 Summary)

Because this is too short to have as its own chapter...

A first inference problem. Author explores a particle decay problem of finding $\Pr\left[\lambda \mid \{x\}\right]$ where λ is the characteristic decay length and $\{x\}$ is our collection of observed decay distances. Plotting the likelihood $\Pr\left[\{x\} \mid \lambda\right]$ *as a function of* λ for any given $x \in \{x\}$ shows each has a *peak* value. The kicker is the interpretation: if each measurement $x \in \{x\}$ is independent, the total likelihood is the product of all the individual likelihoods, which can be interpreted as updating/narrowing the interval $[\lambda_a, \lambda_b]$ within which $\Pr\left[\{x\} \mid \lambda\right]$ (as a function of λ) peaks. In the words of the author's mentor:

> *what you know about λ after the data arrive is what you knew before ($Pr\left[\lambda\right]$) and what the data told you ($Pr\left[\{x\} \mid \lambda\right]$)*

We update our beliefs regarding the distribution of λ as we collect data.

Lessons learned from problems.

(3.8) The classic Monty Hall problem. **Be careful defining probabilities after collecting data**. My blunder: when using Bayes'theorem to get the probability of the prize being behind door $i \in \{1, 2\}$ after the host opens door 3, I failed to take into account that we chose door 1 while I was computing the evidence (the marginal probability that the host opened door 3)[192]. Quite embarrassing.

(3.9) Monty Hall problem, but an earthquake opens door 3. Although I correctly answered that, in this case, both hypotheses (the prize being behind door 1 or door 2) are equiprobable, I still failed to account for the subtle fact that the earthquake could've opened *multiple* doors. The lesson here is **always write down the probability of everything**, which just so happens to be suggested by the solution for this problem, too.

So, why did I still get the answer correct? The reason is because enumeration of probabilities wasn't necessary at all, you just needed to realize that the likelihood for the two remaining hypotheses ($\mathcal{H}_1$ and $\mathcal{H}_2$) were the *same* – the probability of observing the earthquake open door 3 and the prize not being revealed was the same for the case of the prize being behind door 1 or door 2. So maybe the real lesson here is **determine whether calculations are even needed in order to solve the given problem**, which luckily I've had drilled in my head for years from studying physics.

(3.15) Another biased coin variant. One of the best examples I've seen for favoring Bayesian methods over frequentist methods. Also, made use of the beta function:

$$\int_0^1 p^x (1-p)^y \mathrm{d}p = \frac{\Gamma(x+1)\Gamma(y+1)}{\Gamma(x+y+2)} = B(x+1, y+1) \tag{833}$$

where B is the **beta function**, which is defined by the LHS.

[192] Note that, while important to recognize and understand, I could've avoided this pitfall entirely by just ignoring the evidence during calculations and normalizing after, since the evidence can be determined solely by the normalization constraint.

The Source Coding Theorem (Ch. 4)

 Written by Brandon McKinzie

Overview. We will be examining the two following assertions:

1. The Shannon information content is a sensible measure of the information content of a given outcome $x = a_i$:

$$h(x = a_i) \triangleq \lg \frac{1}{p_i} \tag{834}$$

2. The entropy of an ensemble X is a sensible measure of the ensemble's average information content.

$$H(X) = \sum_i p_i \lg \frac{1}{p_i} \tag{835}$$

The weighing problem. We are instructed to ponder the following problem.

> *You're given 12 balls, all equal in weight except for one that is either heavier or lighter. You're also given a two-pan balance. Your task is to determine which ball is the odd ball, and in as few uses of the balance as possible. Note: each use of the balance must place the same number of balls on either side.*

An interesting observation is to consider the number of possible outcomes of the weighing process. Each outcome can be one of three possibilities: equal weight, left heavier, or left lighter. After N such weighings, the number of unique possible weighing result sequences is 3^N. Note that there are $12 \times 2 = 24$ unique final answers for our task (identifying which is the odd ball, and whether it is heavier or lighter). Therefore, since we are seeking a procedure to identify which of the 24 options is the correct option with 100% accuracy, we require our weighing procedure to take on *at least* 24 unique possible results. Since $N = 3$ weighings corresponds to $3^N = 27$ possible outcomes, $N = 3$ is a *lower bound* on the number of weighings our approach will involve. It is impossible to guarantee a correct answer for $N < 3$ weighings[193].

Things I didn't consider until reading the solution:

- It's actually *not* optimal, upon observing both sides equal, to subsequently use only the balls not involved in that measurement. My initial reaction to this was "why? we already know the oddball is not any of the balls just measured, since the outcome was equal."

[193] Finally, it should be clear that, regardless of our approach, the final weighing will involve 2 balls, since we have to identify which is the oddball AND whether it is heavier/lighter AND the number of balls on the left of the scale must be the same as the right of the scale for every weighing.

The response to this reaction is: "yes, *exactly*, and we must use that information to be able to discern in the future whether, e.g., a measurement of "left side heavier" means the oddball is on the left and heavy, or if it's on the right and light – *it's useful to know that a given side of the scale does not contain the oddball before a measurement.*

- More generally, it's also not optimal to greedily search for solutions that eliminate the highest number of possibilities *in any given single step.* Another way of thinking about this is that it's undesirable for the ith measurement outcome to cause any of the 3 possible measurement outcomes to be impossible at the next stage.
- I focused a disproportionate amount of thought on handling the equal-weight measurement outcome, for whatever reason. I probably would've arrived at the solution faster if I'd actually thought about how my strategies would've handled some outcome being "left heavier" and *then considered what that strategy would put on the scale at the next step*, where the italics denote what would've illuminated the fatal flaw in all my approaches.

Guessing Games. What's the smallest number of yes/no questions needed to identify an integer x between 0 and 63? Although it was obvious to me that the solution is to successively halve the possible values of x, I found it interesting that **you can write down the list of questions independent of the answers at each step** using a basic application of modular arithmetic. In other words, you can specify the full decision tree of N nodes with just $\lg N$ questions. Nice. Also, recognize that the Shannon information content for any single outcome is $\lg \frac{1}{0.5} = 1$ bit, and thus the total Shannon information content (for our predefined 6 questions) is 6 bits, which is not-coincidentally the number of possible values that x could be before we ask any questions.

In general, if an outcome x has Shannon information content $h(x)$ number of bits, I like to interpret that as "learning the result x eliminates $2^{h(x)}$ possibilities for the final result." The battleship example follows this interpretation well. Stated another way (in the author's words):

> *The Shannon information content can be intimately connected to the size of a file that encodes the outcomes of a random experiment.*

Data Compression. A **lossy compressor** compresses some files, but maps some files to the *same* encoding. We introduce a parameter δ that describes the risk (aggressiveness of our compression) we are taking with a given compression method: δ is the probability that there will be no name for an outcome[194] x.

The smallest δ-sufficient subset

If S_δ is the smallest subset of $\mathcal{A}_X$ satisfying

$$\Pr\left[x \in S_\delta\right] \geq 1 - \delta \tag{836}$$

then S_δ is the smallest δ-sufficient subset. It can be constructed by ranking the elements of $\mathcal{A}_X$ in order of decreasing probability and adding successive elements starting from the most probable elements until the total probability is $\geq (1-\delta)$.

- **Raw bit content** of X: $H_0(X) \triangleq \lg|\mathcal{A}_X|$. A lower bound for the number of binary questions that are always *guaranteed* to identify an outcome from the ensemble X – it simply maps each outcome to a constant-length binary string.
- **Essential bit content** of X: $H_\delta(X) \triangleq \lg|S_\delta|$. A compression code can be made by assigning a binary string of H_δ bits to each element of the smallest sufficient subset.

Finally, we can now state **Shannon's source coding theorem**: Let X be an ensemble for the random variable x with entropy $H(X) = H$ bits, and let X^N denote a sequence of identically distributed (but not necessarily independent[195]) of random variables/ensembles, $(X_1, X_2, \ldots, X_N)$.

$$(\exists N_0 \in \mathbb{Z}^+)(\forall N > N_0): \qquad \left|\frac{1}{N}H_\delta(X^N) - H\right| < \epsilon \qquad (0 < \delta < 1)\ (\epsilon > 0) \tag{837}$$

which, in English, can be read: *N i.i.d. random variables each with entropy $H(X)$ can be compressed into more than $NH(X)$ bits with negligible risk of information loss, as $N \to \infty$; conversely if they're compressed into fewer than $NH(X)$ bits it is virtually certain that information will be lost.*

[194] More specifically, if there is some subset, $\{a\}$, of unique values that x can take on but our compression method discards/ignores, then we say $\delta = \sum_i p(x = a_i)$.

[195] Actually, before the actual theorem statement, the author mentions we are now concerned with "string of N i.i.d. random variables from a single ensemble X." It's probably fair to assume this is true for the quantities in the theorem, but I'm leaving this note here as a reminder.

Typicality. The reason that large N in equation 837 corresponds to larger potential for better compression is that the subset of likely results for a string of outcomes becomes more and more concentrated relative to the number of possible sequences as N increases[196]. I just realized this is for the same fundamental reasons that entropy exists in thermodynamics – *there are just more ways to exist in a high entropy state than otherwise.* The author showed $\binom{N}{r}$ as a function of r (the number of 1s in the N-bit string). For large N, this becomes almost comically concentrated near the center (like a delta function at $N/2$) – see footnote for more details[197].

This motivates the notion of **typicality** for [a string of length N from] an arbitrary ensemble X with alphabet $\mathcal{A}_X$. For large N, we expect to find $p_i N$ occurrences of the outcome $x = a_i$. Hence the probability of such a string, and its information content, is roughly[198]

$$\Pr\left[\boldsymbol{x}\right]_{typ} = \Pr\left[x_1\right]\Pr\left[x_2\right]\cdots\Pr\left[x_N\right] \simeq p_1^{(p_1 N)} p_2^{(p_2 N)} \cdots p_I^{(p_I N)} \tag{838}$$

$$h(\boldsymbol{x})_{typ} = \lg \frac{1}{\Pr\left[\boldsymbol{x}\right]_{typ}} \simeq N \sum_{i=1}^{I} p_i \lg \frac{1}{p_i} = NH(X) \tag{839}$$

Accordingly, we define the typical elements (strings of length N) of $\mathcal{A}_X^N$ to be those elements that have probability close to 2^{-NH}. We introduce a parameter β that defines what we mean by "close," and define the set of typical elements as the **typical set** $T_{N\beta}$:

$$T_{N\beta} \triangleq \left\{ \boldsymbol{x} \in \mathcal{A}_X^N : \left| \frac{1}{N} \lg \frac{1}{P(\boldsymbol{x})} - H \right| < \beta \right\} \tag{840}$$

It turns out that whatever value of β we choose, the $T_{N\beta}$ contains almost all the probability as N increases.

[196] The author gave an example for a sequence of bits with probability of any given bit being 1 as 0.1. He showed how, although the *average* number of 1s in a sequence of N bits grew as $\mathcal{O}(N)$, the standard deviation of that average only grew as $\sqrt{N}$.

[197] The probability of getting a string with r 1s follows a binomial distribution with mean Np_1 and standard deviation $\sqrt{Np_1(1-p_1)}$. This results in an increasingly narrower distribution $P(r)$ for larger N.

[198] We appear to be assuming that each outcome x in the string $\boldsymbol{x}$ are i.i.d. (CONFIRMED)

Proving the Source Coding Theorem.

- **Setup**. We will make use of the following:
  - **Chebyshev's Inequalities**:

$$\Pr\left[x \geq \alpha\right] \leq \frac{\mathbb{E}\left[x\right]}{\alpha} \qquad \text{and} \qquad \Pr\left[(x - \mathbb{E}\left[x\right])^2 \geq \alpha\right] \leq \frac{\mathrm{Var}\left[x\right]}{\alpha} \tag{841}$$

    where α is a positive real number, and x is assumed non-negative in the first inequality[199].
  - **Weak Law of Large Numbers** (WLLN): Consider a sample $h_1, \ldots, h_N$ of N independent RVs all with common mean $\bar{h}$ and common variance σ_h^2. Let $x = \frac{1}{N}\sum_{n=1}^{N} h_n$ be their average. Then

$$\Pr\left[(x - \bar{h})^2 \geq \alpha\right] \leq \frac{\sigma_h^2}{\alpha N} \tag{842}$$

    which can be easily derived from Chebyshev's inequalities.
- **Proving 'asymptotic equipartition' principle**, i.e. that an outcome $\boldsymbol{x}$ is almost certain to belong to the typical set, approaching probability 1 for large enough N. It is a simple application of the WLLN to the random variable

$$\frac{1}{N}\lg\frac{1}{\Pr\left[\boldsymbol{x}\right]} = \frac{1}{N}\sum_{n=1}^{N}\lg\frac{1}{x_n} = \frac{1}{N}\sum_{n=1}^{N} h(x_n) \tag{843}$$

  where $\mathbb{E}\left[h(x_n)\right] = H(X)$ for all terms in the summation. Observe, then, that the definition of the typical set given in equation 840 (squaring both sides) has the same form as the definition for the WLLN. Plugging in and rearranging yields

$$\Pr\left[\boldsymbol{x} \in T_{N\beta}\right] \geq 1 - \frac{\sigma^2}{\beta^2 N} \tag{844}$$

  where $\sigma^2 \equiv \mathrm{Var}\left[\lg\frac{1}{P(x_n)}\right]$. This proves the asymptotic equipartition principle. It will also be useful to recognize that for any $\boldsymbol{x}$ in the typical set, we can rearrange equation 840 to obtain

$$2^{-N(H+\beta)} < \Pr\left[\boldsymbol{x}\right] < 2^{-N(H-\beta)} \tag{845}$$

- **Proof of SCT Part I**. Want to show that $\frac{1}{N}H_\delta(X^N) < H + \epsilon$. **TODO**
- **Proof of SCT Part II**. Want to show that $\frac{1}{N}H_\delta(X^N) > H - \epsilon$. **TODO**

[199] Notice how the two inequalities are technically the same.

Questions & Answers. Collection of my questions as I read through the chapter, which I answered upon completing it.

- **Q**: Why isn't the essential bit content of a string of N i.i.d. variables NH when $\delta = 0$?
  - **A**: I'm not entirely sure how to answer this still, but it seems the question is confused. First off, the essential bit content approaches the raw bit content as δ decreases to 0: $H_\delta \to H_0$ as $\delta \to 0$. It's important to notice that both H_δ and H_0 define an entropy where all members (S_δ^N for H_δ; $\mathcal{A}_X^N$ for H_0) are *equiprobable*. I remember asking this question wondering "what is the significance of $H_\delta(X^N)$ approaching $NH(X)$ (not a typo!) for tiny δ". The answer: for larger N, more of the probability mass is concentrated in a relatively smaller region, *with elements of that region being roughly equiprobable*. The last part is what I didn't initially realize – that **allowing for tiny δ combined with large N essentially makes it so that $S_\delta^N \approx T_{N\beta}$.**
- **Q**: Why aren't the most probable elements necessarily in the typical set?
  - **A**: In the limit of $N \to \infty$, *they are*, since in that limit, all elements are in the typical set and they're equiprobable. However, in essentially any real case, we can imagine that some elements will be too unlikely to be found within the typical set, which necessarily requires that there exist elements with probability too high to be in the typical set. Remember that the typical set is basically *defined* such that all elements have probability within the range given in equation 845.

Monte Carlo Methods (Ch. 29)

 Written by Brandon McKinzie

Overview. The aims of Monte Carlo methods are to solve one or both of the following:

1. Generate samples $\{\boldsymbol{x}^{(r)}\}_{r=1}^{R}$ from a given probability distribution $\Pr[\boldsymbol{x}]$.
2. Estimate expectations of function under $\Pr[\boldsymbol{x}]$, for example

$$\boldsymbol{\Phi} = \mathbb{E}_{\boldsymbol{x}\sim\Pr[\boldsymbol{x}]}[\phi(\boldsymbol{x})] \equiv \int \mathrm{d}^N\boldsymbol{x}\Pr[\boldsymbol{x}]\,\phi(\boldsymbol{x}) \tag{846}$$

   where it's assumed that $\Pr[\boldsymbol{x}]$ is sufficiently complex that we can't evaluate such expectations by exact methods.

Note that we can concentrate on the first problem (sampling), since we can use it to solve the second problem (estimating an expectation) by using the random samples $\{\boldsymbol{x}^{(r)}\}_{r=1}^{R}$ to give the estimator

$$\hat{\boldsymbol{\Phi}} \equiv \frac{1}{R}\sum_r \phi(\boldsymbol{x}^{(r)}) \tag{847}$$

where

$$\mathbb{E}\left[\hat{\boldsymbol{\Phi}}\right] = \frac{1}{R}\sum_r \mathbb{E}_{\boldsymbol{x}^{(r)}\sim\Pr[\boldsymbol{x}]}\left[\phi(\boldsymbol{x}^{(r)})\right] = \boldsymbol{\Phi} \tag{848}$$

$$\lim_{R\to\infty}\mathrm{Var}\left[\hat{\boldsymbol{\Phi}}\right] = \lim_{R\to\infty}\frac{\sum_r\left(\phi(\boldsymbol{x}^{(r)})-\boldsymbol{\Phi}\right)^2}{R-1} = \frac{\sigma^2}{R} \equiv \frac{1}{R}\int \mathrm{d}^N\boldsymbol{x}\Pr[\boldsymbol{x}]\left(\phi(\boldsymbol{x})-\boldsymbol{\Phi}\right)^2 \tag{849}$$

Importance Sampling. A generalization of [naively] uniformly sampling $\boldsymbol{x}$ in order to approximate equation 846. We assume henceforth that we are able to evaluate (for now, a 1-D) $\Pr[x]$ at any point x at least within a multiplicative constant; thus we can evaluate a function $P^*(x)$ such that $P(x) = P^*(x)/Z$. We assume we have some simpler Q(x), called the **sampler density**, from which we *can* generate samples from and evaluate up to a multiplicative constant, $Q(x) = Q^*(x)/Z_Q$. We construct an approximation for our estimator in equation 847 via sampling from $Q(x)$ and computing:

$$\hat{\boldsymbol{\Phi}} = \frac{\sum_r w_r\phi(x^{(r)})}{\sum_r w_r} \qquad \text{where} \qquad w_r \equiv \frac{P^*(x^{(r)})}{Q^*(x^{(r)})} \tag{850}$$

Rejection Sampling. In addition to the assumptions in importance sampling, we now also assume that we know the value of a constanct c such that

$$\forall x: \quad cQ^*(x) > P^*(x) \tag{851}$$

1. Generate sample x from proposal density $Q(x)$, and evaluate $cQ^*(x)$.
2. Generate a uniformly distributed random variable u from the interval $[0, cQ^*(x)]$.
3. If $u > P^*(x)$, then reject x; else, accept x and add it to our set of samples $\{x^{(r)}\}$.

Metropolis-Hastings Method. Proposal density Q now depends on the current state $x^{(t)}$.

1. Sample tentative next state x' from proposal density $Q(x'; x^{(t)})$.
2. Compute:

$$a = \frac{P^*(x')}{P^*(x^{(t)})} \frac{Q(x^{(t)}; x')}{Q(x'; x^{(t)})} \tag{852}$$

3. If $a \geq 1$, the new state is accepted. Otherwise, the new state is accepted with probability a.
4. If accepted, we set $x^{(t+1)} = x'$. If rejected, we set $x^{(t+1)} = x^{(t)}$.

Variational Methods (Ch. 33)

 Written by Brandon McKinzie

Probability Distributions in Statistical Physics[200]. Consider the common situation below, where the state vector $\boldsymbol{x} \in \{-1, +1\}^N$:

$$\Pr\left[\boldsymbol{x} \mid \beta, \boldsymbol{J}\right] = \frac{e^{-\beta E(\boldsymbol{x};\boldsymbol{J})}}{Z(\beta, \boldsymbol{J})} \tag{853}$$

$$E(\boldsymbol{x};\boldsymbol{J}) = -\frac{1}{2}\sum_{m,n} J_{mn} x_m x_n - \sum_n h_n x_n \tag{854}$$

$$Z(\beta, \boldsymbol{J}) = \sum_{\boldsymbol{x}} e^{-\beta E(\boldsymbol{x};\boldsymbol{J})} \tag{855}$$

Note that evaluating $E(\boldsymbol{x};\boldsymbol{J})$ for a given $\boldsymbol{x}$ takes polynomial time in the number of spins N, and evaluating Z is $\mathcal{O}(2^N)$. Variational free energy minimization is a method for approximating the complex distribution $P(\boldsymbol{x})$ by a simpler ensemble $Q(\boldsymbol{x};\boldsymbol{\theta})$ parameterized by adjustable $\boldsymbol{\theta}$.

Variational Free Energy. How do we find/evaluate Q? The objective function chosen to measure the quality of the approximation is the **variational free energy**, $\widetilde{F}(\boldsymbol{\theta})$:

$$\beta\widetilde{F}(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta}) \ln \frac{Q(\boldsymbol{x};\boldsymbol{\theta})}{e^{-\beta E(\boldsymbol{x};\boldsymbol{J})}} \tag{856}$$

$$= \beta\mathbb{E}_{\boldsymbol{x}\sim Q}\left[E(\boldsymbol{x};\boldsymbol{J})\right] - H(Q) \tag{857}$$

$$= D_{KL}\left(Q \mid\mid P\right) + \beta F \tag{858}$$

where $F \triangleq -\ln Z(\beta, J)$ is the true free energy. It's not immediately clear why this approximation Q is more tractable than P – for that we turn to an example below.

[200] Yay!

Variational Free Energy Minimization for Spin Systems. For the system with energy function given in equation 854, consider the *separable* approximating distribution,

$$Q(\boldsymbol{x};\boldsymbol{a}) = \frac{e^{\sum_n a_n x_n}}{Z_Q} = \frac{\prod_n e^{a_n x_n}}{\sum_{\boldsymbol{x}'} \prod_{n'} e^{a_n x'_{n'}}} = \frac{\prod_n e^{a_n x_n}}{\sum_{x'_1} e^{a_1 x'_1} \sum_{x'_2} e^{a_2 x'_2} \cdots \sum_{x'_n} e^{a_n x'_n}} \tag{859}$$

To compute $\widetilde{F}$, we must compute the mean energy and entropy under Q.

→ **Mean Energy**. Given our definition of Q, and the fact that each $x_n = \pm 1$, the mean value of any x_n is $\bar{x} = \tanh(a_n) = 2q_n - 1$, where $q_n \equiv Q(x_n = 1)$.

$$\mathbb{E}_{\boldsymbol{x}\sim Q}\left[E(\boldsymbol{x};\boldsymbol{J})\right] = \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a}) \left[-\frac{1}{2}\sum_{m,n} J_{mn} x_m x_n - \sum_n h_n x_n\right] \tag{860}$$

$$= -\frac{1}{2}\sum_{m,n} J_{mn}\bar{x}_m\bar{x}_n - \sum_n h_n \bar{x}_n \tag{861}$$

→ **Entropy**. Recall that if $Q(\boldsymbol{x};\boldsymbol{a}) = \prod_n Q(x_n;a_n)$ (i.e. Q is separable), that $H(\boldsymbol{x}) = \sum_n H(x_n)$, so we have

$$H_Q(\boldsymbol{x}) = \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a}) \ln \frac{1}{Q(\boldsymbol{x};\boldsymbol{a})} \tag{862}$$

$$= \sum_n Q(x_n = 1; a_n) \ln \frac{1}{Q(x_n = 1; a_n)} + Q(x_n = 0; a_n) \ln \frac{1}{Q(x_n = 0; a_n)} \tag{863}$$

$$= \sum_n q_n \ln \frac{1}{q_n} + (1 - q_n) \ln \frac{1}{1 - q_n} \tag{864}$$

So the variational free energy is given by

$$\beta\widetilde{F}(\boldsymbol{a}) = \beta\mathbb{E}_{\boldsymbol{x}\sim Q}\left[E(\boldsymbol{x};\boldsymbol{J})\right] - H_Q(\boldsymbol{x}) \tag{865}$$

$$= \beta\left(-\frac{1}{2}\sum_{m,n} J_{mn}\bar{x}_m\bar{x}_n - \sum_n h_n\bar{x}_n\right) - \left(\sum_n q_n \ln\frac{1}{q_n} + (1-q_n)\ln\frac{1}{1-q_n}\right) \tag{866}$$

Remember, our goal is to find parameters $\boldsymbol{a}$ that minimize $\widetilde{F}(a)$:

$$\beta\frac{\partial\widetilde{F}}{\partial a_m} = 2\left(\frac{\partial q_m}{\partial a_m}\right)\left[-\beta\left(\sum_n J_{mn}\bar{x}_n + h_m\right) + a_m\right] \tag{867}$$

which, when set to zero, yields

$$a_m \leftarrow \beta\left(\sum_n J_{mn}\bar{x}_n + h_m\right) \tag{868}$$

$$\bar{x}_n = \tanh(a_n) \tag{869}$$

Machine Learning: A Probabilistic Perspective

Contents

9.1 Probability (Ch. 2) 289

9.2 Generative Models for Discrete Data (Ch. 3) 291

9.2.1 Exercises 295

9.3 Gaussian Models (Ch. 4) 296

9.4 Linear Regression (Ch. 7) 300

9.5 Generalized Linear Models and the Exponential Family (Ch. 9) 303

9.6 Mixture Models and the EM Algorithm (Ch. 11) 306

9.7 Markov and Hidden Markov Models (Ch. 17) 307

9.8 Undirected Graphical Models (Ch. 19) 308

Kevin P. Murphy (2012). Probability.
Machine Learning: A Probabilistic Perspective.

Continuous Random Variables (2.2.5). Let X be a continuous RV. We usually want to know the probability that X lies in the interval $a \leq X \leq b$, which is given by

$$p(a < X \leq b) = p(X \leq b) - p(X \leq a) \tag{870}$$

Define the **cumulative distribution function** (cdf) $F(q) \triangleq p(X \leq q)$, and the **probability density function** (pdf) $f(x) \triangleq \frac{d}{dx}F(x)$.

Transformation of Random Variables (2.6). In what follows, we have some $x \sim p_x$ and $y = f(x)$. How should we think about $p_y(y)$? For discrete RV x, we just sum up the probability mass for all x such that $f(x) = y$,

$$p_y(y) = \sum_{x:f(x)=y} p_x(x) \tag{871}$$

If $\mathbf{x}$ is *continuous*, we must instead work with the cdf,

$$P_y(y) \triangleq P(Y \leq y) = P(f(X) \leq y) = P(X \in \{x \mid f(x) \leq y\}) \tag{872}$$

If f is monotonic (and hence invertible), then we can derive the pdf $p_y(y)$ from the pdf $p_x(x)$ by taking derivatives as follows:

$$p_y(y) \triangleq \frac{d}{dy}P_y(y) = \frac{d}{dy}P_x(f^{-1}(y)) = \frac{dx}{dy}p_x(x) \tag{873}$$

and, since we only work with integrals over densities (i.e. overall sign does not matter), it is convention to take the absolute value of $\frac{dx}{dy}$ in the above formula. In the multivariate case, the Jacobian matrix $[\boldsymbol{J}_{\boldsymbol{y}\to\boldsymbol{x}}]_{i,j} \triangleq \frac{\partial x_i}{\partial y_j}$ is used:

$$p_y(\boldsymbol{y}) = p_x(\boldsymbol{x})\left|\det \boldsymbol{J}_{\boldsymbol{y}\to\boldsymbol{x}}\right| \tag{874}$$

Central Limit Theorem (2.6.3). Consider N i.i.d. RVs each with arbitrary pdf $p(x_i)$, mean μ, and covariance σ^2. Let $S_N = \sum_i X_i$ denote the sum over the N RVs. The **central limit theorem** states that

$$\lim_{N\to\infty} S_N \sim \mathcal{N}(N\mu, N\sigma^2) \tag{875}$$

or equivalently

$$\lim_{N\to\infty} \sqrt{N}(\bar{X} - \mu) \sim \mathcal{N}(0, \sigma^2) \tag{876}$$

Generative Models for Discrete Data (Ch. 3)

Written by Brandon McKinzie

Kevin P. Murphy (2012). Generative Models for Discrete Data.
Machine Learning: A Probabilistic Perspective.

Bayesian Concept Learning (3.1). The interesting notion of learning a *concept* $\mathcal{C}$, such as "all prime numbers", by only seeing positive examples $x \in \mathcal{C}$. How should we approach predicting whether a new test case $\widetilde{x}$ belongs to the concept $\mathcal{C}$? Well, what we're actually doing is as follows: Given an initial **hypothesis space** $\mathcal{H}$ of concepts, we collect data $\mathcal{D}$ that gradually narrows the down the subset of $\mathcal{H}$ consistent with our data. We also need to address how we (humans) will weigh certain hypotheses differently even if they are both entirely consistent with the evidence. The Bayesian approach can be summarized as follows (no particular order):

- **Likelihood**. Probability of observing $\mathcal{D}$ given a particular hypothesis h. In the simple but illustrative case where the data is sampled from a uniform distribution over the *extension*[201] of a concept (a.k.a. the *strong sampling assumption*). The probability here of sampling N items independently under hypothesis h is

$$p(\mathcal{D} \mid h) = \left[\frac{1}{|h|}\right]^N \tag{877}$$

  which elucidates how *the model favors the smallest hypothesis space consistent with the data*[202].
- **Prior**. Using just the likelihood can mean we fall prey to contrived/overfitting hypotheses that basically just enumerate the data. The prior $p(h)$ allows us to specify properties about how the learned hypothesis ought to look.
- **Posterior**. This is just the likelihood times the prior, normalized [to one]. In general, as we collect more and more data, the posterior tends toward a Dirac measure peaked at the MAP estimate:

$$p(h \mid \mathcal{D}) \to \delta_{\hat{h}^{MAP}}(h) \qquad \text{where} \tag{878}$$

$$\hat{h}^{MAP} = \arg\max_h p(h \mid \mathcal{D}) = \arg\max p(\mathcal{D} \mid h)p(h) = \arg\max_h \left[\log p(\mathcal{D} \mid h) + \log p(h)\right] \tag{879}$$

[201] The extension of a concept is just the set of numbers that belong to it.
[202] A result commonly known as **Occam's razor** or the **size principle**.

The Beta-Binomial Model (3.3). In the previous section we considered inferring some discrete distribution over integers; now we will turn to a continuous version. Consider the problem of inferring the probability that a coin shows up heads, given a series of observed coin tosses.

- **Likelihood**. As should be familiar, we'll model the outcome of each toss $X_i \in \{1, 0\}$ indicating heads or tails with $X_i \sim \text{Ber}(\theta)$. Assuming the tosses are i.i.d, this gives us $p(\mathcal{D} \mid \theta) \propto \theta^{N_1}(1-\theta)^{N_0}$, where N_1 and N_0 are the **sufficient statistics** of the data, since $p(\theta \mid \mathcal{D})$ can be entirely modeled as $p(\theta \mid N_1, N_0)$.
- **Prior**. We technically just need a prior $p(\theta)$ with support over the interval $[0, 1]$, but it would be convenient if the prior had the same form as the likelihood, i.e. $p(\theta) \propto \theta^{\gamma_1}(1-\theta)^{\gamma_2}$ for some hyperparameters γ_1 and γ_2 This is satisfied by the **Beta distribution**[203] This would also result in the posterior having the same form as the prior, meaning the prior is a **conjugate prior** for the corresponding likelihood.
- **Posterior**. As mentioned, the posterior corresponding to a Bernoulli/binomial likelihood with a beta prior is itself a beta distribution:

$$p(\theta \mid \mathcal{D}) \propto \text{Bin}(N_1 \mid \theta, N_0 + N_1)\text{Beta}(\theta \mid a, b) \propto \text{Beta}(\theta \mid N_1 + a, N_0 + b) \tag{880}$$

  By either reading off from a table or deriving via calculus, we can obtain the following properties for our Beta posterior:

$$\textbf{[mode]} \qquad \hat{\theta}_{MAP} = \frac{a + N_1 - 1}{a + b + N - 2} \tag{881}$$

$$\textbf{[mean]} \qquad \bar{\theta} = \frac{a + N_1}{a + b + N} = \lambda m_1 + (1 - \lambda)\hat{\theta}_{MLE} \tag{882}$$

  where $m_1 = \frac{a}{a+b}$ is the prior mean. The last form captures the notion that the posterior is a compromise between what we previously believed and what the data is telling us.

The Dirichlet-Multinomial Model (3.4). We now generalize further to inferring the probability that a die with K sides comes up as face k.

- **Likelihood**. As before, we assume a specific observed sequence $\mathcal{D}$ of N dice roles. Assuming the data is i.i.d., the likelihood has the form

$$p(\mathcal{D} \mid \boldsymbol{\theta}) = \prod_{k=1}^{K} \theta_k^{N_k} \tag{883}$$

  which is the likelihood for the multinomial model up to an irrelevant constant factor.
- **Prior**. We'd like to find a conjugate prior for our likelihood, and we need it to have support over the K-dimensional *probability simplex*[204]. The Dirichlet distribution satisfies

[203] You may be wondering, why not the Bernoulli distribution? It trivially has the same form as the Bernoulli distribution, eh? Then, pause and actually think about what you're saying for five seconds. You want to model a prior on θ with a Bernoulli distribution? You do realize that the support for a Bernoulli is in $k \in \{0, 1\}$. It's the opposite domain entirely. We want something that "looks" like a Bernoulli but is a distribution over θ, NOT the value(s) of x.

[204] The K-dimensional probability simplex is the $(K-1)$th dimensional simplex determined by the unit vectors $e_1, \ldots, e_K \in \mathbb{R}^K$, i.e. the set of vectors $\boldsymbol{x}$ such that $x_i \geq 0$ and $\sum_i x_i = 1$.

both of these and is defined as

$$\mathrm{Dir}(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \tag{884}$$

where $\boldsymbol{\theta} \in S_K$ is a built-in assumption.

- **Posterior**. By construction, this will also be Dirichlet. Note that to derive the MAP estimator we must enforce the constraint that $\sum_k \theta_k = 1$, which can be done with a **Lagrange multiplier**. The constrained objective (the Lagrangian) is

$$\ell(\boldsymbol{\theta}, \lambda) = \sum_k N_k \log \theta_k + \sum_k (\alpha_k - 1) \log \theta_k + \lambda \left(1 - \sum_k \theta_k\right) \tag{885}$$

To get $\hat{\theta}_{MAP}$, we'd take derivatives w.r.t. λ and each θ_k, do some substitutions and solve.

Example: Language Models with Bag of Words

Given a sequence of words, predict the most likely next word. Assume that the ith word $X_i \in \{1, \ldots, K\}$ (where K is the size of our vocab) is sampled indep from all others using a Cat($\boldsymbol{\theta}$) (multinoulli) distribution. This is the BoW model.

My attempt: We need to derive the form of posterior predictive $p(X_{i+1} \mid X_1, \ldots, X_i)$ where $\boldsymbol{\theta} \in S_K$. First, I know that the posterior is

$$p(\boldsymbol{\theta} \mid X_1, \ldots, X_i) \propto \mathrm{Dir}(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) P(X_1, \ldots, X_i \mid \boldsymbol{\theta}) = \mathrm{Dir}(\boldsymbol{\theta} \mid \alpha_1 + N_1, \ldots, \alpha_K + N_K) \tag{886}$$

and so I can derive the posterior predictive in the usual way, while also exploiting the fact that all $X_i \perp X_j$,

$$\begin{aligned}
p(X_{i+1} = k \mid X_1, \ldots, X_i) &= \int p(X_{i+1} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid X_1, \ldots, X_i) \mathrm{d}\boldsymbol{\theta} && (887)\\
&= \int \theta_k p(\theta_k, \boldsymbol{\theta}_{-k} \mid X_1, \ldots, X_i) \mathrm{d}\theta_k \mathrm{d}\boldsymbol{\theta}_{-k} && (888)\\
&= \int \theta_k p(\theta_k \mid X_1, \ldots, X_i) \mathrm{d}\theta_k && (889)\\
&= \mathbb{E}\left[\theta_k \mid X_1, \ldots, X_i\right] && (890)\\
&= \frac{\alpha_k + N_k}{\sum_j \alpha_j + N_j} && (891)
\end{aligned}$$

which also shows another nice example of Bayesian smoothing.

Naive Bayes Classifiers (3.5). For classifying vectors of discrete-valued features, $\mathbf{x} \in \{1, \ldots, K\}^D$. Assumes features are conditionally independent given the class label:

$$p(\boldsymbol{x} \mid y = c, \boldsymbol{\theta}) = \prod_{j=1}^{D} p(x_j \mid y = c, \theta_{j,c}) \tag{892}$$

with parameters $\boldsymbol{\theta} \in \mathbb{R}^{D \times |\mathcal{Y}|}$[205]. We can model the individual class-conditional densities with the multinoulli distribution. If we were modeling real-valued features, we could instead use a Gaussian distribution.

- **MLE fitting**. Our log-likelihood is

$$\log p(\mathcal{D} \mid \boldsymbol{\theta}) = \sum_c N_c \log \pi_c + \sum_i^N \sum_j^D \log p(x_j^{(i)} \mid y^{(i)}, \theta_{j,y^{(i)}}) \tag{893}$$

  where $\pi_c = p(y = c)$ are the class priors[206]. We see that we can optimize the class priors separately from the others, and that they have the same form as the multinomial likelihood we used in the last section. We already know that the MLE for these are $\hat{\pi}_c = N_c/N$ (remember this involves using a Lagrangian). Let's assume next that we're working in the case of binary features ($x_j \mid c \sim \text{Ber}(\theta_{j,c})$). This results in MLE estimates $\hat{\theta}_{j,c} = N_{j,c}/N_c$.

[205] You could also generalize this to having some number of params N associated with each pairwise (j, c). It's also important to recognize that this is the first model of this chapter where the input $\boldsymbol{x}$ is a *vector*, and thus introduces pairwise parameters.

[206] We only see the class prior parameters here because they appear in the likelihood of generative classifiers, since $p(x, y) = p(y)p(x \mid y)$. We don't see the priors for θ that aren't class priors because MLE is only concerned with maximizing the likelihood, not the posterior (which would contain those priors).

Exercises

MLE for uniform distribution (3.8)

The density function for the uniform distribution centered on 0 with width 2a is

$$p(x) = \frac{1}{2a}\mathbb{1}\{x \in [-a, a]\}$$

Remember this is for continuous x, and $p(x)$ is interpreted as the derivative of the corresponding CDF $P(x)$.

---

a. Given data $x_1, \ldots, x_n$, find $\hat{a}_{MLE}$. So there are a few ways of doing this. We can get the answer pretty quick with intuition, and not-so-quick by grinding through the math. **Quick-n-easy**: If you were paying attention to the chapter, you'd instantly remember that MLE is all about finding the smallest hypothesis space consistent with the data. It should then be obvious that $\hat{a}_{MLE} = \max |x_i|$. **Slightly more rigorous**. We can also solve a constrained optimization problem, with constraint that $a \geq \max |x_i|$, since we must require our solution to assign nonzero probability to any of our observations.

$$\hat{a}_{MLE} = \arg\max_a \log p(x_1, \ldots, x_n \mid a) + \lambda(a - \max |x_i|) \tag{894}$$

The rest is mechanical: (1) take deriv wrt λ, (2) deriv wrt a and set to zero, (3) solve for a as a function of λ, (4) solve for λ by substituting previous step's results into contstraint, yielding that $\lambda \leq n/|x_{max}|$, (5) plug result for λ into result from step 3 to obtain result that $a \geq x_{max}$. In the limit of many samples, the first term becomes more important in the optimization, which decreases as a increases, and so we choose the lowest value of a that satisfies the constraints: $a := x_{max}$.
b. What probability would the model assign to a new data point x_{n+1} using $\hat{a}$. We are obviously meant to trivially answer that it will assign the density with $a = \hat{a}$. However, I take objection to this question, since it makes no sense to evaluate a density at a single point $p(x)$.
c. Do you see any problem with this approach? Yes, it extremely overfits to the data. We'll assign zero probabilities to any points outside the interval of observations, and the same probability to anything in that interval. We should do a more Bayesian approach instead.

Gaussian Models (Ch. 4)

Kevin P. Murphy (2012). Gaussian Models.
Machine Learning: A Probabilistic Perspective.

Basics (4.1). I'll be filling in the gaps that the book leaves out in its derivations, as a way of reviewing the relevant linear algebra/calculus/etc. For notation's sake, here how the author writes the MVN in D dimensions:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \triangleq \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \tag{895}$$

To get a better understanding, let's inspect the eigendecomposition of $\boldsymbol{\Sigma}$. We know that $\boldsymbol{\Sigma}$ is positive definite[207], and therefore the eigendecomposition $\boldsymbol{\Sigma} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$ exists, where $\boldsymbol{U}$ is an orthonormal matrix of eigenvectors. By the invertible matrix theorem, we therefore know that $\boldsymbol{U}$ is invertible. Since $\boldsymbol{\Sigma}$ is p.d., it's eigenvalues are all positive, and thus $\boldsymbol{\Lambda}$ is also invertible. We can then apply the basic definition for an invertible matrix to write

$$\boldsymbol{\Sigma}^{-1} = \boldsymbol{U}\boldsymbol{\Lambda}^{-1}\boldsymbol{U}^T = \sum_{i=1}^{D} \frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^T \tag{896}$$

Remember, an orthonormal matrix satisfies $U^T = U^{-1}$.

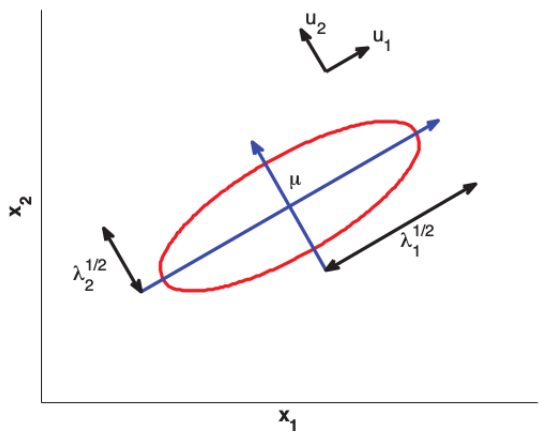
where $\boldsymbol{u}_i$ is the ith eigenvector and ith *column* of $\boldsymbol{U}$. We can use this to rewrite

$$(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) = (\boldsymbol{x}-\boldsymbol{\mu})^T\left(\sum_i^D \frac{1}{\lambda_i}\boldsymbol{u}_i\boldsymbol{u}_i^T\right)(\boldsymbol{x}-\boldsymbol{\mu}) \tag{897}$$

$$= \sum_i^D \frac{1}{\lambda_i}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{u}_i\boldsymbol{u}_i^T(\boldsymbol{x}-\boldsymbol{\mu})^T \tag{898}$$

$$= \sum_i^D \frac{1}{\lambda_i}y_i^2 \tag{899}$$

[207] We know this because all covariance matrices of any random vector X are symmetric p.s.d., and the additional requirement that Σ^{-1} exists means that it is p.d.



where $y_i \triangleq \boldsymbol{u}_i^T(\boldsymbol{x} - \boldsymbol{\mu})$. The fascinating insight is recalling that the equation for an ellipse in 2D is

$$\frac{y_1^2}{\lambda_1} + \frac{y_2^2}{\lambda_2} = 1 \tag{900}$$

Hence we see that the contours of equal probability density of a Gaussian lie along ellipses, as illustrated above. The eigenvectors determine the orientation of the ellipse, and the eigenvalues determine how elongated it is.

Maximum Entropy Derivation of the Gaussian (4.1.4). Recall that the exponential family can be derived as the family of distributions $p(\boldsymbol{x})$ that maximizes $H(p)$ subject to constraints that the moments of p match some set of empirical moments F_k specified by us. It turns out that *the MVN is the distribution with maximum entropy subject to having a specified mean and covariance*. Consider the zero-mean MVN and its entropy[208]:

$$p(\boldsymbol{x}) = \frac{1}{Z} \exp\left\{-\frac{1}{2}\boldsymbol{x}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{x}\right\} \tag{901}$$

$$h(p) = \frac{1}{2} \ln\left[(2\pi e)^D \det \boldsymbol{\Sigma}\right] \tag{902}$$

Let $p = \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ above and let $q(\boldsymbol{x})$ be any density satisfying $\int q(\boldsymbol{x}) x_i x_j \mathrm{d}\boldsymbol{x} = \Sigma_{ij}$. The result is based on the fact that $h(q)$ must be less than or equal to $h(p)$. This can be shown by evaluating $D_{KL}(q||p)$ and recalling that D_{KL} is always greater than or equal to zero.

[208] For derivation, see this wonderful answer on stackexchange.

Gaussian Discriminant Analysis (4.2). With generative classifiers, it is common to define the class-conditional density as a MVN,

$$p(\boldsymbol{x} \mid y{=}c, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{903}$$

which results in a technique called (Gaussian) **discriminant**[209] **analysis**. If Σ_c is diagonal, this is just a form of naive Bayes[210]. We classify some feature vector $\boldsymbol{x}$ using the decision rule:

$$\hat{y}(\boldsymbol{x}) = \arg\max_c \left[\log p(y{=}c \mid \boldsymbol{\pi}) + \log p(\boldsymbol{x} \mid y{=}c, \boldsymbol{\theta})\right] \tag{904}$$

If we have uniform prior over classes, we can classify a new test vector as follows[211]

$$\hat{y}(\boldsymbol{x}) = \arg\min_c (\boldsymbol{x} - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_c) \tag{905}$$

Linear Discriminant Analysis (LDA). The special case where all covariance matrices are the same, $\boldsymbol{\Sigma}_c = \boldsymbol{\Sigma}$. Now the quadratic term $\boldsymbol{x}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{x}$ is independent of c and thus is not important for classification. Instead we end up with the much simpler,

$$p(y{=}c \mid \boldsymbol{x}, \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\beta}_c^T \boldsymbol{x} + \gamma_c}}{\sum_{c'} e^{\boldsymbol{\beta}_{c'}^T \boldsymbol{x} + \gamma_{c'}}} = \mathcal{S}(\boldsymbol{\eta})_c \tag{906}$$

$$\boldsymbol{\beta}_c := \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_c \tag{907}$$

$$\gamma_c := -\frac{1}{2} \boldsymbol{\mu}_c \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_c + \log \pi_c \tag{908}$$

where $\mathcal{S}$ is the familiar **softmax**. **TODO**: come back and compare/contrast LDA with multinomial logistic regression after reading chapter 8.

Inference in Jointly Gaussian Distributions (4.3). Suppose $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2)$ is jointly Gaussian with parameters

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix} \tag{909}$$

Then, the marginals are given by

$$p(\boldsymbol{x}_1) = \mathcal{N}(\boldsymbol{x}_1 \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \tag{910}$$

$$p(\boldsymbol{x}_2) = \mathcal{N}(\boldsymbol{x}_2 \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \tag{911}$$

and the posterior conditional is given by

[209] Don't confuse the word "discriminant" for "discriminative" – we are still in a generative model setting. See 8.6 for details on the distinction.

[210] This is easy to show. If diagonal, then $p(\boldsymbol{x} \mid y)$ factorizes. We know the ith item in the product corresponds to $p(x_i \mid c)$ by considering how simple it is to compute marginals for Gaussians with diagonal Σ.

[211] **TODO**: is the author correct here in leaving out the Σ_c term? We are not assuming equal covariance matrices, so I'm not sure why he can omit that term here. Typo?

$$
\begin{aligned}
p(\boldsymbol{x}_1 \mid \boldsymbol{x}_2) &= \mathcal{N}(\boldsymbol{x}_1 \mid \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\
\text{where} \qquad \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2) \\
\boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1}
\end{aligned} \tag{4.69}
$$

where $\boldsymbol{\Lambda} := \boldsymbol{\Sigma}^{-1}$. Notice that the conditional covariance is a *constant* matrix independent of $\boldsymbol{x}_2$.

Linear Gaussian Systems (4.4). Suppose we have a hidden variable $\boldsymbol{x}$ and a noisy observation of it $\boldsymbol{y}$. Let's assume we have the following prior and likelihood:

$$
\begin{aligned}
p(\boldsymbol{x}) &= \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x) \\
p(\boldsymbol{y} \mid \boldsymbol{x}) &= \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{\Sigma}_y)
\end{aligned} \tag{4.124}
$$

which is an example of a **linear Gaussian system** $\boldsymbol{x} \to \boldsymbol{y}$.

Linear Regression (Ch. 7)

Kevin P. Murphy (2012). Linear Regression.
Machine Learning: A Probabilistic Perspective.

Model Specification (7.2). The linear regression model is a model of the form

$$p(y \mid \boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}(y \mid \boldsymbol{w}^T \boldsymbol{x}, \sigma^2) \tag{912}$$

Maximum Likelihood Estimation (least squares) (7.3). Most commonly, we'll estimate the parameters by computing the MLE, defined by

$$\hat{\boldsymbol{\theta}} \triangleq \arg\max_{\theta} \log p(\mathcal{D} \mid \boldsymbol{\theta}) \tag{913}$$

$$= \arg\min_{\theta} \left[-\log p(\mathcal{D} \mid \boldsymbol{\theta})\right] \tag{914}$$

$$= -\frac{1}{2\sigma^2} RSS(\boldsymbol{w}) - \frac{N}{2}\log(2\pi\sigma^2) \tag{915}$$

$$RSS(\boldsymbol{w}) \triangleq \sum_{i=1}^{N} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 \tag{916}$$

where $RSS(\boldsymbol{w})$ is the **residual sum of squares**.

Derivation of the MLE (7.3.1). We'll now denote the negative log likelihood as $\text{NLL}(\boldsymbol{w})$ and drop constant terms that are irrelevant for the optimization task.

$$\text{NLL}(\boldsymbol{w}) = \frac{1}{2}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}||_2^2 = \frac{1}{2}\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w} - \boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{y}) \tag{917}$$

where $$\boldsymbol{X}^T\boldsymbol{X} = \sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{x}_i^T \tag{918}$$

$$\boldsymbol{X}^T\boldsymbol{y} = \sum_{i=1}^{N} \boldsymbol{x}_i y_i \tag{919}$$

And the optimal $\hat{\boldsymbol{w}}_{OLS}$ be found by taking the gradient, setting to zero, and solving for $\boldsymbol{w}$ as usual:

$$\nabla \text{NLL}(\boldsymbol{w}) = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y} \tag{920}$$

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \qquad \textbf{[normal eq.]} \tag{921}$$

$$\hat{\boldsymbol{w}}_{OLS} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{922}$$

Geometric Interpretation (7.3.2). Use the column-vector representation of $\boldsymbol{X} \in \mathbb{R}^{N\times D}$. Our prediction can then be written

$$\hat{y} = \boldsymbol{X}\boldsymbol{w} = w_1\tilde{\boldsymbol{x}}_1 + \cdots + w_D\tilde{\boldsymbol{x}}_D \tag{923}$$

i.e. a linear combination of the D column vectors $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^N$. Crucially, observe that this means $\hat{y} \in \text{span}(\{\tilde{\boldsymbol{x}}\}_i^D)$ no matter what (a hard constraint by definition of our model). So, how do you minimize the residual norm $||\boldsymbol{y} - \hat{\boldsymbol{y}}||$ given that $\hat{\boldsymbol{y}}$ is restricted to a particular subspace? You require $\boldsymbol{y} - \hat{\boldsymbol{y}}$ to be orthogonal to that subspace, of course[212]! Formally, this means $\tilde{\boldsymbol{x}}_j^T(\boldsymbol{y} - \hat{\boldsymbol{y}}) = 0$, for all $1 \leq j \leq D$. Equivalently,

$$\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) = \boldsymbol{0} \quad \implies \quad \hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{924}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{P}\boldsymbol{y} \tag{925}$$

$$\text{where} \qquad \boldsymbol{P} \triangleq \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T \tag{926}$$

Robust Linear Regression (7.4). Gaussians sensitive to outliers, since their log-likelihood penalizes deviations quadratically[213]. One way to achieve **robustness** to outliers is to instead use a distribution with **heavy tails**, such that they still allow for higher likelihoods of outliers, but they don't need to shift the whole distribution around to accommodate for them. One popular choice is the **Laplace distribution**,

$$p(y \mid \boldsymbol{x}, \boldsymbol{w}, b) = \text{Lap}(y \mid \boldsymbol{w}^T\boldsymbol{x}, b) \triangleq \frac{1}{2b}\exp\left\{-\frac{1}{b}|y - \boldsymbol{w}^T\boldsymbol{x}|\right\} \tag{927}$$

$$NLL(\boldsymbol{w}) = \sum_i |y_i - \boldsymbol{w}^T\boldsymbol{x}_i| \tag{928}$$

Goal: convert NLL to a form easier to optimize (linear). Let $r_i \triangleq y_i - \boldsymbol{w}^T\boldsymbol{x}_i$ be the i'th residual. The following steps show how we can convert this into a **linear program**:

$$r_i \triangleq r_i^+ - r_i^- \qquad (r_i^+ \geq 0)(r_i^- \geq 0) \tag{929}$$

$$\min_{\boldsymbol{w},\boldsymbol{r}^+,\boldsymbol{r}^-} \sum_i (r_i^+ + r_i^-) \qquad \text{s.t.} \qquad \boldsymbol{w}^T\boldsymbol{x}_i + r_i^+ - r_i^- = y_i \tag{930}$$

$$\min_{\boldsymbol{\theta}} \boldsymbol{f}^T\boldsymbol{\theta} \qquad \text{s.t.} \qquad \boldsymbol{A}\boldsymbol{\theta} \leq \boldsymbol{b},\ \boldsymbol{A}_{eq}\boldsymbol{\theta} = \boldsymbol{b}_{eq},\ \boldsymbol{1} \leq \boldsymbol{\theta} \leq \boldsymbol{u} \tag{931}$$

[212] Consider that $\boldsymbol{y} - \hat{\boldsymbol{y}}$ points *from* our prediction (which is in the constraint subspace) $\hat{\boldsymbol{y}}$ *to* the true $\boldsymbol{y}$ that we want to get closest to. Intuitively that means $\hat{\boldsymbol{y}}$ is looking "straight out" at $\boldsymbol{y}$, in a direction orthogonal to the subspace that $\hat{\boldsymbol{y}}$ lives in.

[213] In other words, outliers initially get huge loss values, and the distribution shifts toward them to minimize loss (undesirably).

where the last equation is the **standard form** of a LP.

Ridge Regression (7.5). We know that MLE can overfit by essentially memorizing the data. If, for example, we model 21 points with a degree-14 polynomial[214], we get many large positive and negative numbers for our learned coefficients, which allow the curve to wiggle in just the right way to almost perfectly interpolate the data – **this is why we often regularize weights to have low absolute value**. This encourages smoother/less-wiggly curves. One way to do this is by using a zero-mean Gaussian prior on our weights:

$$p(\boldsymbol{w}) = \prod_j \mathcal{N}(w_j \mid 0, \tau^2) \tag{932}$$

This makes our MAP estimation problem and solution take the form

$$J(\boldsymbol{w}) = \frac{1}{N}\sum_i^N \Big(y_i - \boldsymbol{w}^T\boldsymbol{x}_i - w_0\Big) + \lambda||\boldsymbol{w}||_2^2 \tag{933}$$

$$\hat{\boldsymbol{w}}_{ridge} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_D)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{934}$$

Note that w_0 is NOT regularized.

[214] To review polynomials, search "Lagrange interpolation" in your CS 70 notes.

Generalized Linear Models and the Exponential Family (Ch. 9)

Written by Brandon McKinzie

Kevin P. Murphy (2012). Generalized Linear Models and the Exponential Family.
Machine Learning: A Probabilistic Perspective.

The Exponential Family (9.2). Why is the exponential family important?

- It's the only family with **finite-sized sufficient statistics**[215].
- It's the only family for which **conjugate priors exist**.
- It makes the **least set of assumptions** subject to some user-chosen constraints.
- It's at the core of GLMs and variational inference.

*A pdf or pmf $p(\boldsymbol{x} \mid \boldsymbol{\theta})$, for $\boldsymbol{x} \in \mathcal{X}^m$ and $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^d$, is said to be in the **exponential family** if it's of the form*

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} h(\boldsymbol{x}) \exp\{\boldsymbol{\theta}^T \boldsymbol{\phi}(\boldsymbol{x})\} = h(\boldsymbol{x}) \exp\left\{\boldsymbol{\theta}^T \boldsymbol{\phi}(\boldsymbol{x}) - A(\boldsymbol{\theta})\right\} \tag{935}$$

$$Z(\boldsymbol{\theta}) = \int_{\mathcal{X}^m} h(\boldsymbol{x}) \exp\{\boldsymbol{\theta}^T \boldsymbol{\phi}(\boldsymbol{x})\} \tag{936}$$

$$A(\boldsymbol{\theta}) = \log Z(\boldsymbol{\theta}) \tag{937}$$

$h(\boldsymbol{x})$ is a scaling constant, often 1.

*where $\boldsymbol{\theta}$ are the **natural (canonical) parameters**[216], and $\boldsymbol{\phi}(\boldsymbol{x})$ are the **sufficient statistics**.*

Below are some (quick/condensed) examples showing the first couple steps in rewriting familiar distributions in exponential family form:

$$\textbf{[Bernoulli]} \quad \mathrm{Ber}(x \mid \mu) = \mu^x (1-\mu)^{1-x} = \exp\{x \log \mu + (1-x)\log(1-\mu)\} \tag{938}$$

$$\textbf{[Multinoulli]} \quad \mathrm{Cat}(x \mid \boldsymbol{\mu}) = \prod_k^K \mu_k^{x_k} = \exp\left\{\sum_k^{K-1} x_k \log\left(\mu_k/\mu_K\right) + \log \mu_K\right\} \tag{939}$$

[215] Given certain regularity conditions.

[216] We often generalize this with $\boldsymbol{\eta}(\boldsymbol{\theta})$, which maps whatever params $\boldsymbol{\theta}$ we've chosen to the canonical params $\boldsymbol{\eta}(\boldsymbol{\theta})$.

MLE for the Exponential Family (9.2.4). The likelihood takes the form

$$p(\mathcal{D} \mid \boldsymbol{\theta}) = \left[\prod_i^N h(\boldsymbol{x}^{(i)})\right] g(\boldsymbol{\theta})^N \exp\left\{\boldsymbol{\eta}(\boldsymbol{\theta})^T \boldsymbol{\phi}(\mathcal{D})\right\} \tag{940}$$

It appears that we are denoting $1/Z$ with g now.

$$\boldsymbol{\phi}(\mathcal{D}) = \sum_i^N \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) = \begin{bmatrix} \sum_i \phi_1 & \cdots & \sum_i \phi_K \end{bmatrix} \tag{941}$$

where I've denoted $\sum_{i=1}^N \phi_k(\boldsymbol{x}^{(i)})$ as simply $\sum_i \phi_k$. The **Pitman-Koopman-Darmois theorem** states, given certain regularity conditions/constraints[217], that the exponential family is the only family with *finite sufficient statistics* (dimensionality independent of the size of the data set). For example, in the above formula, we have $K+1$ sufficient statistics (+1 since we need to know the value of N).

Consider a canonical[218] exponential family which also sets $h(\cdot) = 1$. The log-likelihood is

$$p(\mathcal{D} \mid \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathcal{D}) - NA(\boldsymbol{\theta}) \tag{942}$$

Since $-A(\boldsymbol{\theta})$ is concave[219] and the other term is linear in $\boldsymbol{\theta}$, the log-likelihood is concave and thus has a global maximum.

Maximum Entropy Derivation of the Exponential Family (9.2.6). Suppose we don't know which distribution p to use, but we do know the expected values of certain features or functions:

$$F_k \triangleq \mathbb{E}_{\boldsymbol{x} \sim p(\mathbf{x})}\left[f_k(\boldsymbol{x})\right] = \sum_{\boldsymbol{x}} f_k(\boldsymbol{x}) p(\boldsymbol{x}) \tag{943}$$

The principle of **maximum entropy** or **maxent** says we should pick the distribution with maximum entropy, subject to the constraints that the moments of the distribution match the empirical moments of the specified functions $f_k(\boldsymbol{x})$. Treating p as a fixed length vector (i.e. assuming $\boldsymbol{x}$ is discrete), we can take the derivative of our Lagrangian (entropy in units of nats with constraints) w.r.t. each "element" $p_x = p(\boldsymbol{x})$ to find the optimal distribution.

$$J(p, \boldsymbol{\lambda}) = H(p) + \lambda_0 \left(1 - \sum_{\boldsymbol{x}} p(\boldsymbol{x})\right) + \sum_k \lambda_k \left(F_k - \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f_k(\boldsymbol{x})\right) \tag{944}$$

$$\frac{\partial J}{\partial p(\boldsymbol{x})} = -1 - \log p(\boldsymbol{x}) - \lambda_0 - \sum_k \lambda_k f_k(\boldsymbol{x}) \tag{945}$$

[217] The wording is weird here. We mean "out of all families/distributions that *already* satisfy certain constraints that must be met, the exponential family is the only...". For example, the uniform distribution has finite statistics and is not in the exponential family, but it does not meet the constraint that its support must be independent of the parameters, so it's outside the scope of the theorem.

[218] Defined as those which satisfy $\boldsymbol{\eta}(\boldsymbol{\theta}) = \boldsymbol{\theta}$.

[219] We know $-A$ is concave because A is convex. We know A is convex because $\nabla^2 A$ is positive definite. Remember that any twice-differentiable multivariate function f is convex IFF its Hessian is pd for all $\boldsymbol{\theta}$. See sec 7.3.3 and 9.2.3 for more.

Setting this derivative to zero yields

$$p(\boldsymbol{x}) = \frac{1}{Z} \exp\left\{ -\sum_k \lambda_k f_k(\boldsymbol{x}) \right\} \tag{946}$$

Thus the maxent distribution $p(\boldsymbol{x})$ has the form of the exponential family, a.k.a. the **Gibbs distribution**.

Mixture Models and the EM Algorithm (Ch. 11)

Written by Brandon McKinzie

Kevin P. Murphy (2012). Mixture Models and the EM Algorithm.
Machine Learning: A Probabilistic Perspective.

Latent Variable Models (LSMs) (11.1). In this chapter, we explored directed GMs that have hidden variables.

Mixture Models (11.2). The simplest case is where the hidden variables $z_i \in \{1, \ldots, K\}$ represent a discrete latent state. We use discrete prior $p(z_i) = \text{Cat}(\boldsymbol{\pi}) = \pi_i$, and likelihood $p(\boldsymbol{x}_i \mid z{=}k) = p_k(\boldsymbol{x}_i)$. A **mixture model** is defined by

$$p(\boldsymbol{x}_i \mid \boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k p_k(\boldsymbol{x}_i \mid \boldsymbol{\theta}) \tag{947}$$

We call p_k the kth **base distribution**.

The EM Algorithm (11.4).

Kevin P. Murphy (2012). Markov and Hidden Markov Models.
Machine Learning: A Probabilistic Perspective.

The Forwards Algorithm (17.4.2). For computing the filtered[220] marginals $p(z_t \mid \boldsymbol{x}_{\langle 1 \ldots t \rangle})$, where z_t is the hidden state at time t. The algorithm has two steps:

1. **Prediction step**. Compute the *one-step-ahead predictive density* $p(z_t \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle})$,

$$p(z_t{=}j \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle}) = \sum_i p(z_t = j, z_{t-1} = i \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle}) \tag{948}$$

   which will serve as our prior for time t (since it does not take into account observed data at time t).
2. **Update step**. We "update" our beliefs by observing $\boldsymbol{x}_t$,

$$\begin{aligned}
\alpha_t(j) &\triangleq p(z_t{=}j \mid \boldsymbol{x}_{\langle 1 \ldots t \rangle}) && (949)\\
&= \frac{p(z_t{=}j, \boldsymbol{x}_t, \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle})}{p(\boldsymbol{x}_{\langle 1 \ldots t \rangle})} && (950)\\
&= \frac{\cancel{p(\boldsymbol{x}_{\langle 1 \ldots t-1 \rangle})} p(z_t{=}j \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle}) p(\boldsymbol{x}_t \mid z_t{=}j, \cancel{\boldsymbol{x}_{\langle 1 \ldots t-1 \rangle}})}{\cancel{p(\boldsymbol{x}_{\langle 1 \ldots t-1 \rangle})} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle})} && (951)\\
&= \frac{1}{Z_t} p(z_t{=}j \mid \boldsymbol{x}_{\langle 1 \ldots t-1 \rangle}) p(\boldsymbol{x}_t \mid z_t{=}j) && (952)
\end{aligned}$$

Forwards Algorithm (Algorithm 17.1)

We are given transition matrix $T_{i,j} = p(z_t = j \mid z_{t-1} = i)$, evidence vectors $\psi_t(j) = p(\boldsymbol{x}_t \mid z_t{=}j)$, and initial state distribution $\pi(j) = p(z_1 = j)$.

1. First, compute the initial $[\alpha_1, Z_1] = \text{normalize}(\psi_1 \odot \pi)$.
2. For time $2 \leq t \leq T$, compute $[\alpha_t, Z_t] = \text{normalize}(\psi_t \odot \boldsymbol{T}^T \boldsymbol{\alpha}_{t-1})$.
3. Return $\boldsymbol{\alpha}_{\langle 1 \ldots T \rangle}$ and $\log p(\boldsymbol{x}_{\langle 1 \ldots T \rangle}) = \sum_t \log Z_t$.

[220] They're called "filtered" because they use all observations $\boldsymbol{x}_{\langle 1 \ldots t \rangle}$ instead of just $\boldsymbol{x}_t$, which reduces/filters out the noise more.

Kevin P. Murphy (2012). Undirected Graphical Models.
Machine Learning: A Probabilistic Perspective.

Learning (19.5). Consider a MRF in log-linear form over C cliques and its log-likelihood (scaled by 1/N):

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left\{\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\boldsymbol{y})\right\} \tag{953}$$

$$\ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_i \left[\sum_c \boldsymbol{\theta}_c^T \boldsymbol{\phi}_c(\boldsymbol{y}) - \log Z(\boldsymbol{\theta})\right] \tag{954}$$

We know from chapter 9 that this log-likelihood is concave in $\boldsymbol{\theta}$, and that $\frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z = \mathbb{E}\left[\phi_c\right]$. So the gradient of the LL is

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} = \left[\frac{1}{N} \sum_i \boldsymbol{\phi}_c(\boldsymbol{y})\right] - \mathbb{E}_{p(\boldsymbol{y}|\boldsymbol{\theta})}\left[\boldsymbol{\phi}_c(\boldsymbol{y})\right] \tag{955}$$

Convex Optimization

Contents

10.1 Convex Sets (Ch. 2) 310

Boyd and Vandenberghe (2004). Convex Sets.
Convex Optimization.

Lines and line segments

Viewed as a function of $\theta \in \mathbb{R}$, we can express the equation for a ***line*** *in $\mathbb{R}^n$ in the following two ways:*

$$y = \theta x_1 + (1-\theta)x_2 \tag{956}$$
$$y = x_2 + \theta(x_1 - x_2) \tag{957}$$

for some $x_1, x_2 \neq x_1 \in \mathbb{R}^n$. If we restrict $\theta \in [0,1]$, we have a ***line segment****.*

Affine sets

A set $C \subseteq \mathbb{R}^n$ is ***affine*** *if the line (not just segment) through any two distinct points in C also lies in C. More generally, this implies that for any set of points $\{x_1, \ldots, x_k\}$, with each $x_i \in C$, all* ***affine combinations****,*

$$\sum_{i=1}^{k} \theta_i x_i, \quad \textit{where} \quad \sum_i \theta_i = 1 \tag{958}$$

are in C, too. Related terminology:

- ***affine hull*** *of any set $C \subseteq \mathbb{R}^n$, denoted $\mathbf{aff}\, C$, is the set of all affine combinations of points in C.*
- ***affine dimension*** *of a set C is the dimension of its affine hull.*
- ***relative interior*** *of a set C, denoted $\mathbf{relint}\, C$, is its interior*[221] *relative to $\mathbf{aff}\, C$,*

$$\mathbf{relint}\, C \triangleq \{x \in C \mid B(x,r) \cap \mathbf{aff}\, C \subseteq C \textit{ for some } r > 0\} \tag{959}$$

There's a lot of neat things to say here, but I only have space to state the results:

- If C is an affine set and $x_0 \in C$, then

$$V = C - x_0 \triangleq \{x - x_0 \mid x \in C\} \tag{960}$$

  is a *subspace*, i.e. closed under sums and scalar multiplication.

[221] The **interior of a set** $C \subseteq \mathbb{R}^n$, denoted $\mathbf{int}\, C$, is the set of all points interior to C. A point $x \in C$ is interior to C if $\exists \epsilon > 0$ for which *all points* in the set

$$\{y \in R^n \mid ||y - x||_2 \leq \epsilon\}$$

are also in C.

- The solution set of a system of linear equations, $C = \{x \mid Ax = b\}$, is an affine set. The subspace associated with C is the *nullspace* of A.

Convex sets

A set C is ***convex*** *if it contains all* ***convex combinations****,*

$$\sum_{i=1}^{k} \theta_i x_i, \quad \textit{where} \quad \sum_i \theta_i = 1, \ \textit{and } \theta_i \geq 0 \tag{961}$$

Related terminology:

- ***convex hull*** *a set C, denoted* $\mathbf{conv}C$*, is the set of all convex combinations of points in C.*

Cones

A set C is called a ***cone*** *if $(\forall x \in C)(\theta \geq 0)$ we have $\theta x \in C$. A set C is called a* ***convex cone*** *if it contains all* ***conic combinations****,*

$$\sum_{i=1}^{k} \theta_i x_i, \quad \textit{where} \quad \theta_i \geq 0 \tag{962}$$

of points in C. Related terminology:

- ***conic hull*** *of a set C is the set of all conic combinations of points in C.*

Bayesian Data Analysis

Contents

11.1 Probability and Inference (Ch. 1) 313
11.2 Single-Parameter Models (Ch. 2) 315
11.3 Asymptotics and Connections to Non-Bayesian Approaches (Ch. 4) 318
11.4 Gaussian Process Models (Ch. 21) 321

Probability and Inference (Ch. 1)

 Written by Brandon McKinzie

The process of Bayesian Data Analysis can be divided into the following 3 steps:

1. Setting up a *full probability model*.
2. Conditioning on observed data. Calculating the *posterior distribution* over unobserved quantities, given observed data.
3. Evaluating the fit of the model and implications of the posterior.

Notation: In general, we let θ denote unobservable vector quantities or population *parameters* of interest, and y as collected data. This means our *posterior* takes the form $p(\theta \mid y)$, and our *likelihood* takes the form $p(y \mid \theta)$.

Means and Variances of Conditional Distributions.

$$\mathbb{E}[u] = \mathbb{E}_v[\mathbb{E}_u[u \mid v]] \tag{963}$$

$$\mathrm{Var}[u] = \mathbb{E}_v[\mathrm{Var}[u \mid v]] + \mathrm{Var}[\mathbb{E}_u[u \mid v]] \tag{964}$$

Proofs

$$\mathbb{E}[u] = \int \mathrm{d}v \mathrm{Pr}[v] \int \mathrm{d}u \mathrm{Pr}[u \mid v] \tag{965}$$

$$= \mathbb{E}_v[\mathbb{E}_u[u \mid v]] \tag{966}$$

$$\mathrm{Var}[u] = \mathbb{E}\left[u^2\right] - \mathbb{E}[u]^2 \tag{967}$$

$$= \mathbb{E}_v\left[\mathbb{E}_u\left[u^2 \mid v\right]\right] - \mathbb{E}_v[\mathbb{E}_u[u \mid v]]^2 \tag{968}$$

$$= \mathbb{E}_v\left[\mathbb{E}_u\left[u^2 \mid v\right] - \mathbb{E}_u[u \mid v]^2\right] + \mathbb{E}_v\left[\mathbb{E}_u[u \mid v]^2\right] - \mathbb{E}_v[\mathbb{E}_u[u \mid v]]^2 \tag{969}$$

$$= \mathbb{E}_v[\mathrm{Var}[u \mid v]] + \mathrm{Var}[\mathbb{E}_u[u \mid v]] \tag{970}$$

Transformation of Variables.

$$\mathrm{Pr}_v[v] = |J| \mathrm{Pr}_u\left[f^{-1}(v)\right] \tag{971}$$

$J_{i,j} = \frac{\partial u}{\partial v} = \frac{\partial f^{-1}(v)}{\partial v}$

where u and v have the same dimensionality, and $|J|$ is the determinant of the Jacobian of the transformation $u = f^{-1}(v)$. When working with parameters defined on the open unit interval, $(0, 1)$, we often use the logistic transformation:

$$\mathrm{logit}(u) = \log\left(\frac{u}{1-u}\right) \tag{972}$$

$$\mathrm{logit}^{-1}(v) = \log\left(\frac{e^v}{1+e^v}\right) \tag{973}$$

Standard Probability Distributions[222].

Distribution	Notation	Density Function
Beta	$\text{Beta}(\alpha, \beta)$	$p(\theta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$
Inverse-gamma	$\text{Inv-gamma}(\alpha, \beta)$	$p(\theta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\theta^{-(\alpha+1)}e^{-\beta/\theta}$
Normal (univariate)	$\text{N}(\mu, \sigma^2)$	$p(\theta) = \frac{1}{\sqrt{2\pi\sigma}}\exp\big(-\frac{1}{2\sigma^2}(\theta-\mu)^2\big)$
Scaled inverse-χ^2	$\text{Inv-}\chi^2(\nu, s^2)$	$\theta \sim \text{Inv-gamma}(\frac{\nu}{2}, \frac{\nu}{2}s^2)$

[222]The **gamma function**, $\Gamma(x)$, is defined as

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}\mathrm{d}t \tag{974}$$

or simply as $(x-1)!$ if $x \in \mathbb{Z}^+$.

Since we'll be referring to the binomial model frequently, below are the main distributions for reference:

$$p(y \mid \theta) = Bin(y \mid n, \theta) = \binom{n}{y} \theta^y (1-\theta)^{n-y} \tag{975}$$

$$p(\theta \mid y) \propto \theta^y (1-\theta)^{n-y} \tag{976}$$

$$\theta \mid y \sim \text{Beta}(y+1, n-y+1) \tag{977}$$

where y is the number of successes out of n trials. We assume a uniform prior over the interval [0, 1].

Posterior as compromise between data and prior information. Intuitively, the prior and posterior distributions over θ should have some general relationship showing how the process of observing data y updates our distribution on θ. We can use the identities from the previous chapter to see that

$$\mathbb{E}[\theta] = \mathbb{E}_y [\mathbb{E}_\theta [\theta \mid y]] \tag{978}$$

$$\text{Var}[\theta] = \mathbb{E}_y [\text{Var}[\theta \mid y]] + \text{Var}[\mathbb{E}_\theta [\theta \mid y]] \tag{979}$$

where:

→ Equation 978 states the obvious: our prior expectation for θ is the expectation, taken over the distribution of possible data, of the posterior expectation.

→ Equation 979 states: *the posterior variance is on average smaller than the prior variance*, by an amount that depends on the variation [in posterior means] over the distribution of possible data. Stated another way: we can reduce our uncertainty with regard to θ by larger amounts with models whose [expected] posteriors are strongly informed by the data.

Informative Priors. We now discuss some the issues that arise in assigning a prior distribution $p(\theta)$ that reflects substantive information. Instead of using a uniform prior for our binomial model, let's explore the prior $\theta \sim \text{Beta}(\alpha, \beta)$[223]. Now our posterior takes the form

$$\theta \mid y \propto \text{Beta}(\alpha + y, \beta + n - y) \tag{980}$$

The property that the posterior follows the same parametric form as the prior is called **conjugacy**.

[223] Assume we can select reasonable values for α and β.

Conjugacy, families, and sufficient statistics. Formally, if $\mathcal{F}$ is a class of sampling distributions $\{\Pr_i[y \mid \theta]\}$, and $\mathcal{P}$ is a class of prior distributions, $\{\Pr_j[\theta]\}$, then the class $\mathcal{P}$ is **conjugate** for $\mathcal{F}$ if[224]

$$\forall \Pr[\cdot \mid \theta] \in \mathcal{F}, \Pr[\cdot] \in \mathcal{P}: \quad \Pr[\theta \mid y] \in \mathcal{P} \tag{981}$$

Probability distributions that belong to an **exponential family** have natural conjugate prior distributions. The class $\mathcal{F}$ is an exponential family if all its members have the form,

$$\Pr[y_i \mid \theta] = f(y_i) g(\theta) e^{\phi(\theta)^T u(y_i)} \tag{982}$$

$$\Pr[y \mid \theta] = \left(\prod_{i=1}^{n} f(y_i)\right) g(\theta)^n \exp\left(\phi(\theta)^T \sum_{i=1}^{n} u(y_i)\right) \tag{983}$$

$$\propto g(\theta)^n \exp\left(\phi(\theta)^T t(y)\right) \tag{984}$$

where

- $y = (y_1, \ldots, y_n)$ denotes n iid observations.
- $\phi(\theta)$ is called the **natural parameter** of $\mathcal{F}$.
- $t(y) = \sum_{i=1}^{n} u(y_i)$ is said to be a **sufficient statistic** for θ, because the likelihood for θ depends on the data y only through the value of $t(y)$.

Normal distribution with known variance[225]. Consider a single scalar observation y drawn from $N(\theta, \sigma^2)$, where we assume σ^2 is known. The family of conjugate prior densities for the Gaussian likelihood as well as our choice of parameterization are, respectively,

Defining our conjugate prior

$$p(\theta) \propto e^{A\theta^2 + B\theta + C} \tag{986}$$

$$p(\theta) \propto e^{-\frac{1}{2\tau_0^2}(\theta - \mu_0)^2} \tag{987}$$

By definition, this implies that the posterior should also be normal. Indeed, after some basic arithmetic/substitutions, we find

Writing our posterior precision and mean.

$$\Pr[\theta \mid y] \propto \exp\left(-\frac{1}{2\tau_1^2}(\theta - \mu_1)^2\right) \tag{988}$$

$$\mu_1 = \frac{\frac{1}{\tau_0^2}\mu_0 + \frac{1}{\sigma^2} y}{\frac{1}{\tau_0^2} + \frac{1}{\sigma^2}} \quad \text{and} \quad \frac{1}{\tau_1^2} = \frac{1}{\tau_0^2} + \frac{1}{\sigma^2} \tag{989}$$

[224] In English: A class of prior distributions is conjugate for a class of sampling distributions if, for any pair of sampling distribution and prior distribution [from those two respective classes], the associated posterior distribution is *also* in the same class of prior distributions.

[225] The following will be useful to remember:

$$\int_{-\infty}^{\infty} e^{-ax^2 + bx + c} \mathrm{d}x = \frac{\pi}{a} e^{\frac{b^2}{4a} + c} \tag{985}$$

where we see that the posterior **precision** (inverse of variance) equals the prior prior precision plus the data precision. We can see the posterior mean μ_1 expressed as a weighted average of the prior mean and the observed value[226] y, with weights proportional to the precisions.

Normal distribution with unknown variance. Now, we assume the mean θ is known, and the variance σ^2 is unknown. The likelihood for a vector $y = (y_1, \ldots, y_n)$ of n iid observations is

Computing our likelihood for n IID observations.

$$\Pr\left[y \mid \sigma^2\right] \propto (\sigma^2)^{-n/2} \exp\left(-\frac{n}{2\sigma^2} v\right) \tag{990}$$

$$v := \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta)^2 \tag{991}$$

where v is the sufficient statistic. The corresponding conjugate prior density is the inverse-gamma. This and our choice for parameterization (how we define α and β) are, respectively,

Defining our conjugate prior.

$$\Pr\left[\sigma^2\right] \propto (\sigma^2)^{-(\alpha+1)} e^{-\beta/\sigma^2} \tag{992}$$

$$\sigma^2 \sim \text{Inv-}\chi^2(\nu_0, \sigma_0^2) = \text{Inv-gamma}(\tfrac{\nu_0}{2}, \tfrac{\nu_0}{2}\sigma_0^2) \tag{993}$$

All that's left is computing our posterior,

$$p(\sigma^2 \mid y) \propto p(\sigma^2) p(y \mid \sigma^2) \tag{994}$$

$$\sigma^2 \mid y \sim \text{Inv-}\chi^2\left(\nu_0 + 2, \frac{\nu_0\sigma_0^2 + nv}{\nu_0 + n}\right) \tag{995}$$

Jeffrey's Invariance Principle. An approach for defining noninformative prior distributions. Let $\phi = h(\theta)$, where the function h is one-to-one. By transformation of variables,

$$p(\phi) = p(\theta)\left|\frac{d\theta}{d\phi}\right| = p(\theta)|h'(\theta)|^{-1} \tag{996}$$

Jeffrey's principle is to basically take the above equation as a true equivalence – there is no difference between finding $p(\theta)$ and applying the equation above [to get $p(\phi)$] and directly finding $p(\phi)$.

Let $J(\theta)$ denote the **Fisher information** for θ, defined as

$$J(\theta) = \mathbb{E}\left[\left(\frac{d \log \Pr\left[y \mid \theta\right]}{d\theta}\right)^2 \mid \theta\right] = -\mathbb{E}\left[\frac{d^2 \log \Pr\left[y \mid \theta\right]}{d\theta^2} \mid \theta\right] \tag{997}$$

Jeffrey's prior model defines the noninformative prior density as $\Pr\left[\theta\right] \propto \left[J(\theta)\right]^{1/2}$. We can work out that this model is indeed invariant to parameterization[227].

[226] For now, we're considering the single data point case.

[227] Evaluate $J(\phi)$ at $\theta = h^{-1}(\phi)$. You should find that $J(\phi)^{1/2} = J(\theta)^{1/2}|\frac{d\theta}{d\phi}|$

Normal Approximations to the Posterior Distribution. If the posterior $\Pr[\theta \mid y]$ is unimodal and roughly symmetric, it can be convenient to approximate it by a normal distribution. Here we'll consider a quadratic approximation via the Taylor series expansion up to second-order,

$$\log \Pr[\theta \mid y] = \log \Pr\left[\hat{\theta} \mid y\right] + \frac{1}{2}(\theta - \hat{\theta})^T \left[\frac{d^2}{d\theta^2} \log \Pr[\theta \mid y]\right]_{\theta=\hat{\theta}} (\theta - \hat{\theta}) \tag{998}$$

where $\hat{\theta}$ is the posterior mode. The remainder terms of higher order fade in importance relative to the quadratic term when θ is close to $\hat{\theta}$ and n is large. We'd like to cast this into a normal distribution. First, let

$$I(\theta) \triangleq -\frac{d^2}{d\theta^2} \log \Pr[\theta \mid y] \tag{999}$$

which we will refer to as the **observed information**. We can then rewrite our approximation as[228]

$$\Pr[\theta \mid y] \approx \mathcal{N}(\hat{\theta}, [I(\hat{\theta})]^{-1}) \tag{1002}$$

> *Under the normal approximation, the posterior distribution is summarized by its mode, $\hat{\theta}$, and the curvature of the posterior density, $I(\hat{\theta})$; that is, asymptotically, these are sufficient statistics.*

[228] I also found it helpful to explicitly write the substitution after raising eq 998 by power of e (all logs are assumed natural logs)

$$\Pr[\theta \mid y] = e^{\log \Pr\left[\hat{\theta}|y\right] - \frac{1}{2}(\theta-\hat{\theta})^T \left[I(\hat{\theta})\right]^{-1}(\theta-\hat{\theta})} \tag{1000}$$

$$= \Pr\left[\hat{\theta} \mid y\right] e^{-\frac{1}{2}(\theta-\hat{\theta})^T \left[I(\hat{\theta})\right]^{-1}(\theta-\hat{\theta})} \tag{1001}$$

Example. Let $y_1, \ldots, y_n$ be independent observations from $\mathcal{N}(\mu, \sigma^2)$. Define $\theta := (\mu, \log \sigma)$ as the parameters of interest, and assume a uniform prior[229] $\Pr[\theta]$. Recall that (equation 3.2 in textbook)

$$\Pr[\theta = (\mu, \log\sigma) \mid y] \propto \sigma^{-(n+2)} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2\right) \tag{1004}$$

Note that $\sum_{i=1}^{n}(y_i - \bar{y}) = 0$

$$= \sigma^{-(n+2)} \exp\left(-\frac{1}{2\sigma^2}\left[n(\bar{y} - \mu)^2 + \sum_{i=1}^{n}(y_i - \bar{y})^2\right]\right) \tag{1005}$$

$$= \sigma^{-(n+2)} \exp\left(-\frac{1}{2\sigma^2}\left[n(\bar{y} - \mu)^2 + (n-1)s^2\right]\right) \tag{1006}$$

where $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2$ is the sample variance of the y_i's. The sufficient statistics are $\bar{y}$ and s^2. To construct the approximation, we need the second derivatives of the log posterior density[230],

$$\log \Pr[\mu, \log\sigma \mid y] = \text{const} - n\log\sigma - \frac{1}{2\sigma^2}\left(n(\bar{y} - \mu)^2 + (n-1)s^2\right) \tag{1007}$$

in order to compute $I(\hat{\theta})$. After computing first derivatives, we find that the posterior mode is

$$\hat{\theta} = (\hat{\mu}, \log\hat{\sigma}) = \left(\bar{y}, \log\left(\sqrt{\frac{n-1}{n}}s\right)\right) \tag{1008}$$

We then compute second derivatives and evaluate at $\theta = \hat{\theta}$ to obtain $I(\hat{\theta})$. Combining all this into the final result:

$$\Pr[\mu, \log\sigma \mid y] \approx \mathcal{N}(\hat{\theta}, [I(\hat{\theta})]^{-1}) \tag{1009}$$

$$= \mathcal{N}\left(\begin{pmatrix}\bar{y} \\ \log\hat{\sigma}\end{pmatrix}, \begin{pmatrix}\hat{\sigma}^2/n & 0 \\ 0 & 1/(2n)\end{pmatrix}\right) \tag{1010}$$

where $\hat{\sigma}^2/n$ is the variance along the μ dimension, and $1/(2n)$ is the variance along the $\log\sigma$ direction. This example was just meant to illustrate, with a simple case, how we work through constructing the approximate normal distribution.

[229] Recall from Ch 3.2 that the uniform prior on $\mu, \log\sigma$ is

$$\Pr\left[\mu, \sigma^2\right] \propto (\sigma^2)^{-1} \tag{1003}$$

where we continue to assume μ is uniform in $[0, 1]$ for some reason.

[230] I'm not sure why $n + 2$ has seemingly turned into n.

Large-Sample Theory. Asymptotic normality of the posterior distribution: as more data arrives from the same underlying distribution $f(y)$, the posterior distribution of the *parameter vector* θ approaches multivariate normality, even if the true distribution of the data is not within the parametric family under consideration.

Suppose the data are modeled by a parametric family, $\Pr[y \mid \theta]$, with a prior distribution $\Pr[\theta]$. If the true distribution, $f(y)$, is included in the parametric family (i.e. if $\exists \theta_0 : f(y) = \Pr[y \mid \theta_0]$), then it's also true that **consistency** holds[231]: $\Pr[\theta \mid y]$ converges to a point mass at the true parameter value, θ_0 as $n \to \infty$.

[231] So, what if the true $f(y)$ is *not* included in the parametric family? In that case, there is no longer a true value θ_0, but its role in the theoretical result is replaced by a value θ_0 that makes the model distribution $\Pr[y \mid \theta]$ closest to the true distribution $f(y)$, in a technical sense involving the **Kullback-Leibler divergence**.

Gaussian Process Models (Ch. 21)

 Written by Brandon McKinzie

Since I like to begin by motivating what we're going to talk about, and since BDA doesn't really do this, I'm going to start with an excerpt from chapter 15 of Kevin Murphy's book:

> *In supervised learning, we observe some inputs $\boldsymbol{x}_i$ and some outputs y_i. We assume that $y_i = f(\boldsymbol{x}_i)$, for some unknown function f, possibly corrupted by noise. The optimal approach is to infer a distribution over functions given the data, $p(f \mid \boldsymbol{X}, \boldsymbol{y})$, and then to use this to make predictions given new inputs, i.e., to compute*
>
> $$p(y_* \mid \boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) = \int p(y_* \mid f, \boldsymbol{x}_*) p(f \mid \boldsymbol{X}, \boldsymbol{y}) \mathrm{d}f \tag{1011}$$
>
> ***Gaussian Processes*** *or GPs define a prior over functions $p(f)$ which can be converted into a posterior over functions $p(f \mid \boldsymbol{X}, \boldsymbol{y})$ once we've seen some data.*

Gaussian Process Regression (20.1). We write a GP as $\mu \sim \mathrm{GP}(m, k)$ (μ is now taking the place of f from Kevin Murphy's notation), parameterized in terms of a mean function m and a covariance function k. Remember, μ is supposed to represent the predictor function for obtaining output predictions given inputs, $y = \mu(x)$. Instead of making the usual assumption that there is some "best" function μ^* and trying to learn it via fitting a parameterized $\hat{\mu}(\boldsymbol{\theta})$, we are going *full meta*[232] (i.e. full Bayesian) and learning the distribution over predictors.

Apparently, we only need to consider a finite (and arbitrary) set of points $x_1, \ldots, x_n$ to consider when evaluating any given μ. The **GP prior** on μ is defined as

$$\mu(x_1), \ldots, \mu(x_n) \sim \mathcal{N}\left(\{m(x_1), \ldots, m(x_n)\}, K(x_1, \ldots, x_n)\right) \tag{1012}$$

with mean m and covariance K[233]. The covariance function k specifies the covariance between the process at any two points, with K an $n \times n$ covariance matrix with $K_{p,q} = k(x_p, x_q)$. The covariance function controls the smoothness of realizations from the GP[234] and the degree of shrinkage towards the mean.

[232] *What if there are like, a whole space of different predictors, man? Like, what if there is an infinite sea of predictor functions, all with their own unique traits and quirks? Woah.*

[233] Don't confuse the notation – $\mathcal{N}$ uses covariance K as an argument, while GP uses covariance *function* k as an argument.

[234] In English: How similar we expect different samples of μ to look as a function of x. The reason this was weird to think about at first is because I'm used to thinking about covariance/smoothness over x rather than sampled *functions* of x. Meta.

A common choice the squared exponential,

$$k(x, x') = \tau^2 \exp\left(-\frac{|x - x'|^2}{2\ell^2}\right) \tag{1013}$$

where τ controls the magnitude and ℓ the smoothness of the function.

Gaussian Processes for Machine Learning

Contents

12.1 Regression (Ch. 2) 324

Rasmussen and Williams (2006). Regression. *Gaussian Processes for Machine Learning.*

Weight-space view (2.1). We review the standard probabilistic view of linear regression.

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{w} \tag{1014}$$

$$y = f(\boldsymbol{x}) + \varepsilon \quad \text{where} \quad \varepsilon \sim \mathcal{N}(0, \sigma_n^2) \tag{1015}$$

$X \in \mathbb{R}^{d \times n}$

$$\text{[likelihood]} \qquad \boldsymbol{y} \mid X, \boldsymbol{w} \sim \mathcal{N}(X^T \boldsymbol{w}, \sigma_n^2 I) \tag{1016}$$

$$\text{[prior]} \qquad \boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \Sigma_p) \tag{1017}$$

$$\text{[posterior]} \qquad p(\boldsymbol{w} \mid \boldsymbol{y}, X) = \frac{p(\boldsymbol{y} \mid \boldsymbol{w}, X) p(\boldsymbol{w})}{p(\boldsymbol{y} \mid X)} \tag{1018}$$

$$\sim \mathcal{N}(\frac{1}{\sigma_n^2} A^{-1} X \boldsymbol{y}, A^{-1}) \tag{1019}$$

where $A = \sigma_n^{-2} X X^T + \Sigma_p^{-1}$, and we often set $\Sigma_p = I$. When analyzing the contour plots for the likelihood, remember that it is *not* a probability distribution, but rather it's interpreted as a function of $\boldsymbol{w}$, i.e. likelihood($\boldsymbol{w}$) $:= \mathcal{N}(\boldsymbol{y}; X^T \boldsymbol{w}, \sigma_n^2 I)$.

Note that we often use Bayesian techniques without realizing it. For example, what does the following remind you of?

$$\ln p(\boldsymbol{w}) \propto \frac{1}{2} \boldsymbol{w}^T \Sigma_p \boldsymbol{w} \tag{1020}$$

It's the l2 penalty from ridge regression (where typically $\Sigma_p = I$). We can also project the inputs to a higher-dimensional space, often referred to as the *feature space*, by passing them through feature functions $\phi(x)$. As we'll see later (ch 5), GPs actually tell us how to define the basis functions $h_i(x)$ which define the value of $\phi_i(x)$, the ith element of the feature vector. The author then proceeds to give an overview of the kernel trick.

Function-space view (2.2). Instead of inference over parameters $\boldsymbol{w}$, we can equivalently consider inference in function space with a **Gaussian process** (GP), formally defined as

> *A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

A GP is completely specified by its mean function and covariance function. Define mean function $m(\boldsymbol{x})$ and covariance function $k(\boldsymbol{x}, \boldsymbol{x}')$ of a real [Gaussian] process $f(\boldsymbol{x})$ as

$$m(\boldsymbol{x}) = \mathbb{E}_f\left[f(\boldsymbol{x})\right] \tag{1021}$$
$$k(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}_f\left[\left(f(\boldsymbol{x}) - m(\boldsymbol{x})\right)\left(f(\boldsymbol{x}') - m(\boldsymbol{x}')\right)\right] \tag{1022}$$
$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')\right) \tag{1023}$$

$$\mathbb{E}_{\boldsymbol{w}\sim\mathcal{N}(\mathbf{0},\Sigma_p)}\left[f(\boldsymbol{x};\boldsymbol{w})\right] = \phi(\boldsymbol{x})^T\mathbb{E}\left[\boldsymbol{w}\right] = 0 \tag{1024}$$
$$\mathbb{E}_{\boldsymbol{w}\sim\mathcal{N}(\mathbf{0},\Sigma_p)}\left[f(\boldsymbol{x})f(\boldsymbol{x}')\right] = \phi(\boldsymbol{x})^T\mathbb{E}\left[\boldsymbol{w}\boldsymbol{w}^T\right]\phi(\boldsymbol{x}') = \phi(\boldsymbol{x})^T\Sigma_p\phi(\boldsymbol{x}') \tag{1025}$$

A common covariance function is the **squared exponential** (a.k.a. the RBF kernel),

$$k(\boldsymbol{x}_p, \boldsymbol{x}_q) = \mathrm{Cov}\left[f(\boldsymbol{x}_p),\ f(\boldsymbol{x}_q)\right] = \exp(-\frac{1}{2}|\boldsymbol{x}_p - \boldsymbol{x}_q|^2) \tag{1026}$$

where it's important to recognize that, whereas we've usually seen this is the RBF kernel for the purposes of kernel methods on inputs, we are now using it specify the *covariance of the* ***outputs***.

Ok, so how do we sample some functions and plot them? Below is an overview for our current linear regression running example.

1. Choose a number of input points X_*. For our linear regression example, we could set this to `np.arange(-5, 5.1, 0.1)` to get evenly spaced x in $[-5, 5]$ in intervals of 0.1.
2. Write out the covariance matrix defined by $K_{p,q} = k(x_p, x_q)$ using our squared exponential covariance function, for all pairs of inputs.
3. We can now generate samples of function f, represented as a random vector with size equal to the number of inputs $|X_*|$, by sampling from the **GP prior**

$$\boldsymbol{f}_* \sim \mathcal{N}(\mathbf{0}, K(X_*, X_*)) \tag{1027}$$

So far, we've only dealt with the GP *prior*. What do we do when we get labeled training observations? How do we make predictions on unlabeled test data? Well, for the simple case where our observations are noise free[235], that is we know $\{(\boldsymbol{x}_i, f_i) \mid i = 1, \ldots, n\}$, the joint GP prior over the train set inputs X and test set inputs X_* is defined exactly how we did

[235] For example, noise-free linear regression would mean we model y=f(x), implicitly defining $\varepsilon = 0$.

it earlier (zero mean, elementwise evaluation of k). In other words, our GP prior models the train outputs $\boldsymbol{f}$ and test outputs $\boldsymbol{f}_*$ as random vectors sampled via

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{1028}$$

You may be wondering: *why are we talking about sampling from the prior on inputs? We already know the outputs!*, and you'd be correct. The way we obtain our posterior is by restricting our joint prior to only those functions that agree with the observed training data $X, \boldsymbol{f}$, which we can do by simply conditioning on them. Our posterior for sampling test outputs given test inputs X_* is thus

$$\begin{aligned} \boldsymbol{f}_* \mid X_*, X, \boldsymbol{f} \sim \mathcal{N}\Big(& K(X_*,X)K(X,X)^{-1}\boldsymbol{f}, \\ & K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)\Big) \end{aligned} \tag{1029}$$

Blogs

Contents

13.1 Conv Nets: A Modular Perspective 328
13.2 Understanding Convolutions 329
13.3 Deep Reinforcement Learning 331
13.4 Deep Learning for Chatbots (WildML) 333
13.5 Attentional Interfaces – Neural Perspective 335

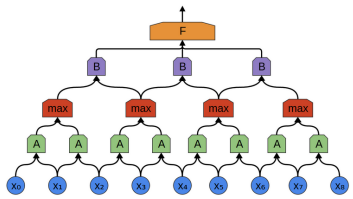
Blogs **December 21, 2016**

Conv Nets: A Modular Perspective

Table of Contents Local *Written by Brandon McKinzie*

From this post on Colah's Blog.

The title is inspired by the following figure. Colah mentions how groups of neurons, like A, that appear in multiple places are sometimes called **modules**, and networks that use them are sometimes called modular neural networks. You can feed the output of one convolutional layer into another. With each layer, the network can detect higher-level, more abstract features.



$\longrightarrow$ Function of the A neurons: compute certain *features*.

$\longrightarrow$ Max pooling layers: kind of "zoom out". They allow later convolutional layers to work on larger sections of the data. They also make us invariant to some very small transformations of the data.

Blogs **December 21, 2016**

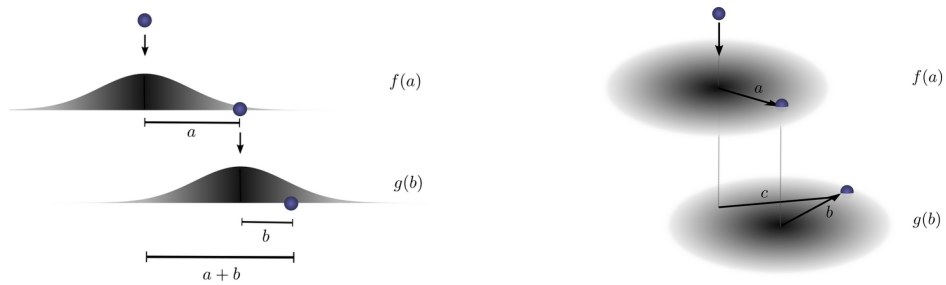
Understanding Convolutions

Table of Contents Local *Written by Brandon McKinzie*

From Colah's Blog.

Ball-Dropping Example. The posed problem:

> Imagine we drop a ball from some height onto the ground, where it only has one dimension of motion. How likely is it that a ball will go a distance c if you drop it and then drop it again from above the point at which it landed?



From basic probability, we know the result is a sum over possible outcomes, constrained by $a + b = c$. It turns out this is actually the definition of the convolution of f and g.

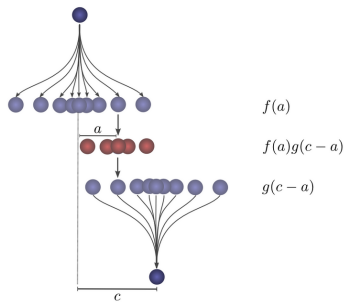
$$\Pr(\mathrm{a} + \mathrm{b} = \mathrm{c}) = \sum_{a+b=c} f(a) \cdot g(b) \tag{1030}$$

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b) \tag{1031}$$

$$= \sum_{a} f(a) \cdot g(c-a) \tag{1032}$$

Visualizing Convolutions. Keeping the same example in the back of our heads, consider a few interesting facts.

- **Flipping directions**. If $f(x)$ yields the probability of landing a distance x away from where it was dropped, what about the probability that it was dropped a distance x from where it *landed*? It is $f(-x)$.
- Above is a visualization of one term in the summation of $(f * g)(c)$. It is meant to show how we can move the bottom around to think about evaluating the convolution for different c values.



0	0	0	0	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

We can relate these ideas to image recognition. Below are two common kernels used to convolve images with.

On the left is a kernel for *blurring* images, accomplished by taking simple averages. On the right is a kernel for *edge detection*, accomplished by taking the difference between two pixels, which will be largest at edges, and essentially zero for similar pixels.

Blogs **December 23, 2016**

Deep Reinforcement Learning

Table of Contents Local *Written by Brandon McKinzie*

Link to tutorial – Part I of "Demystifying deep reinforcement learning."

Reinforcement Learning. Vulnerable to the *credit assignment problem* - i.e. unsure which of the preceding actions was responsible for getting some reward and to what extent. Also need to address the famous *explore-exploit dilemma* when deciding what strategies to use.

Markov Decision Process. Most common method for representing a reinforcement problem. MDPs consist of states, actions, and rewards. Total reward is sum of current (includes previous) and *discounted* future rewards:

$$R_t = r_t \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1} \tag{1033}$$

Q - learning. Define function $Q(s, a)$ to be best possible score at end of game after performing action a in state s; the "quality" of an action from a given state. The recursive definition of Q (for one transition) is given below in the *Bellman equation*.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

and updates are computed with a learning rate α as

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_{t-1}, a_{t-1}) + \alpha \cdot (r + \gamma \max_{a'} Q(s'_{t+1}, a'_{t+1}))$$

Deep Q Network. Deep learning can take deal with issues related to prohibitively large state spaces. The implementation chosen by DeepMind was to represent the Q-function with a neural network, with the states (pixels) as the input and Q-values as output, where the number of output neurons is the number of possible actions from the input state. We can optimize with simple squared loss:

$$L = \frac{1}{2}[\underbrace{r + max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

and our algorithm from some state s becomes

1. **First forward pass** from s to get all predicted Q-values for each possible action. Choose action corresponding to max output, leading to next s'.

2. **Second forward pass** from s' and again compute $\max_{a'} Q(s', a')$.
3. **Set target output** for each action a' from s'. For the action corresponding to max (from step 2) set its target as $r + \gamma \max_{a'} Q(s', a')$, and for all other actions set target to same as originally returned from step 1, making the error 0 for those outputs. (Interpret as update to our guess for the best Q-value, and keep the others the same.)
4. **Update weights** using backprop.

Experience Replay. This the most important trick for helping convergence of Q-values when approximating with non-linear functions. During gameplay all the experience $< s, a, r, s' >$ are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition.

Exploration. One could say that initializing the Q-values randomly and then picking the max is essentially a form of exploitation. However, this type of exploration is *greedy*, which can be tamed/fixed with **ε-greedy exploration**. This incorporates a degree of randomness when choosing next action at *all* time-steps, determined by probability ε that we choose the next action randomly. For example, DeepMind decreases ε over time from 1 to 0.1.

Deep Q-Learning Algorithm.

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Blogs **January 15, 2017**

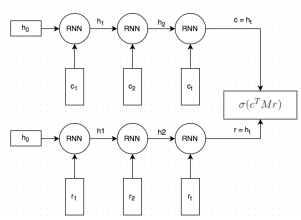
Deep Learning for Chatbots (WildML)

Table of Contents Local *Written by Brandon McKinzie*

Overview.

- **Model**. Implementing a retrieval-based model. Input: conversation/context c. Output: response r.
- **Data**. Ubuntu Dialog Corpus (UDC). 1 million examples of form (context, utterance, label). The label can be 1 (utterance was actual response to the context) or a 0 (utterance chosen randomly). Using NLTK, the data has been . . .
  → **Tokenized**: dividing strings into lists of substrings.
  → **Stemmed**. **IDK**
  → **Lemmatized**. **IDK**
  The test/validation set consists (context, ground-truth utterance, [9 distractors (incorrect utterances)]). The distractors are picked at random[236]

Dual-Encoder LSTM.



1. **Inputs**. Both the context and the response text are split by words, and each word is embedded into a vector and fed into the same RNN.

2. **Prediction**. Multiply the [vector representation ("meaning")] c with param matrix M to predict some response r'.

3. **Evaluation**. Measure similarity of predicted r' to actual r via simple dot product. Feed this into sigmoid to obtain a probability [of r' being the correct response]. Use (binary) cross-entropy for loss function:

$$L = -y \cdot \ln(y') - (1 - y) \cdot \ln(1 - y') \tag{1034}$$

where y' is the predicted probability that r' is correct response r, and $y \in \{0, 1\}$ is the true label for the context-response pair (c, r).

[236] Better example/approach: Google's Smart Reply uses clustering techniques to come up with a set of possible responses.

Data Pre-Processing. Courtesy of WildML, we are given 3 files after preprocessing: train.tfrecords, validation.tfrecords, and test.tfrecords, which use TensorFlow's 'Example' format. Each Example consists of . . .

- context: Sequence of word ids.
- context_len: length of the aforementioned sequence.
- utterance: seq of word ids representing utterance (response).
- utterance_len.
- label: only in training data. 0 or 1.
- distractor_[N]: Only in test/validation. N ranges from 0 to 8. Seq of word ids reppin the distractor utterance.
- distractor_[N]_len.

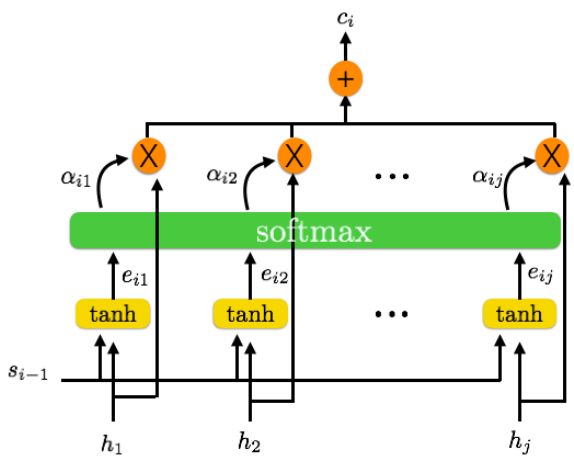
Blogs **April 04, 2017**

Attentional Interfaces – Neural Perspective

Table of Contents Local *Written by Brandon McKinzie*

[Link to article]

Attention Mechanism. Below is a close-up view/diagram of an attention layer. Technically, it only corresponds to a single time step i; we are using the previous decoder state s_{i-1} to compute the ith context vector c_i which will be fed as an input to the decoder for step i.



For convenience, I'll rewrite the familiar equations for computing quantities at some step i.

$$\text{[decoder state]} \qquad s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{1035}$$

$$\text{[context vect]} \qquad c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{1036}$$

$$\text{[annotation weights]} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{1037}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{1038}$$

Now we can see just how simple this really is. Recall that Bahdanau *et al.*, 2015 use the wording: "e_{ij} *is an alignment model which scores how well the inputs around position* j *and the output at position* i *match*." But we can see an example implementation of an alignment model above: the *tanh* function (that's it).

Appendix

Contents

14.1 Common Distributions and Models 337
14.2 Math 338
14.3 Main Tasks in NLP 341
14.4 Misc. Topics 344

Distribution	Density Function	Notation
Bernoulli	$p(x;\theta) = \theta^{\mathbb{1}_{x=1}}(1-\theta)^{\mathbb{1}_{x=0}}$	$X \sim Ber(\theta)$
Binomial	$p(x;n) = \binom{n}{x} p^x (1-p)^{n-x}$	$x \sim \text{Bin}(n,p)$
Chi-Square	$p(\theta) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)}\theta^{\nu/2-1}e^{-\theta/2}, \quad \theta > 0$	$\theta \sim \chi^2_\nu$
Gamma	$p(\theta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\theta^{\alpha-1}e^{-\beta\theta}, \quad \theta > 0$	$\theta \sim \text{Gamma}(\alpha,\beta)$
Inverse-gamma	$p(\theta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\theta^{-\alpha-1}e^{-\beta/\theta}, \quad \theta > 0$	$\theta \sim \text{Inv-gamma}(\alpha,\beta)$
Inverse-chi-square	$p(\theta) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)}\theta^{-\nu/2-1}e^{-\frac{1}{2\theta}}, \quad \theta > 0$	$\theta \sim \text{Inv-}\chi^2_\nu$
Multinomial	$p(x_1,\ldots,x_k;n) = \frac{n!}{\prod_i^k x_i!}\prod_i^k p_i^{x_i}$	

$(\forall n \in \mathbb{Z}^+)\Gamma(n) = (n-1)!$

$\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}\mathrm{d}x$

Logistic Regression. Perhaps the simplest *linear method*[237] for classification is **logistic regression**. Let K be the number of classes that y can take on. The model is defined as

$$\Pr[y=k \mid \boldsymbol{x}] = \frac{\exp(\boldsymbol{\theta}_k^T\boldsymbol{x})}{1+\sum_{\ell=1}^{K-1}\exp(\boldsymbol{\theta}_\ell^T\boldsymbol{x})}, \quad \text{for } 1 \le k \le K-1 \tag{1039}$$

$$\Pr[y=K \mid \boldsymbol{x}] = \frac{1}{1+\sum_{\ell=1}^{K-1}\exp(\boldsymbol{\theta}_\ell^T\boldsymbol{x})} \tag{1040}$$

and we often denote $\Pr[y=k \mid \boldsymbol{x}]$ under the entire set of parameters $\boldsymbol{\theta}$ simply as $p_k(\boldsymbol{x};\boldsymbol{\theta})$ or just $p_k(\boldsymbol{x})$. The decision boundaries are the set of points in the domain of $\boldsymbol{x}$ for which some $p_k(\boldsymbol{x}) = p_{j\neq k}(\boldsymbol{x})$. Equivalently, the model can be specified by $K-1$ log-odds or logit transformations of the form

$$\log\left(\frac{p_i(\boldsymbol{x})}{p_K(\boldsymbol{x})}\right) = \boldsymbol{\theta}_i^T\boldsymbol{x} \quad \text{for } 1 \le i \le K-1 \tag{1041}$$

Also note that the parameter vectors are orthogonal to the K-1 decision boundaries. For any x, x' on the decision boundary defined as the set of points $\{x : p_a(x) = p_b(x)\}$, we know that the vector $x - x'$ is parallel to the decision boundary (by definition), and can derive

$$\frac{p_a(x)}{p_b(x)} = 1 = \exp(\theta_a^T x - \theta_b^T x) \implies \theta_a^T x = \theta_b^T x \tag{1042}$$

$$\therefore \theta_a^T(x-x') = \theta_b^T(x-x') = 0 \tag{1043}$$

and thus θ_a and θ_b are both perpendicular to the decision boundary where $p_a(x) = p_b(x)$.

[237] We say a classification method is *linear* if its **decision boundary is linear**.

Fancy math definitions/concepts for fancy authors who require fancy terminology.

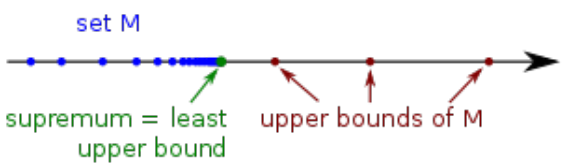
- **Support**. Sloppy definition you'll see in most places: The *set-theoretic support* of a real-valued function $f : X \mapsto \mathbb{R}$ is defined as

$$\text{supp}(f) \triangleq \{x \in X : f(x) \neq 0\}$$

  Note that Wikipedia is surprisingly sloppy with how it defines and/or uses support in various articles. After some digging, I finally found the formal definition for probability and measure theory:

  > *If $X : \Omega \mapsto \mathbb{R}$ is a random variable on $(\Omega, \mathcal{F}, P)$, then the **support** of X is the smallest closed set $R_X \subset \mathbb{R}$ such that $P(X \in R_X) = 1$.*

- **Infimum and Supremum**. The fancy-math way of saying minimum and maximum. Yes, I recognize that these are important in certain (usually rather abstract) settings, but often in ML it is used when sup means exactly the same thing as max, but the authors want to look sophisticated. Here I'll give the formal definition for sup. You have a partially ordered set[238] P, and are for the moment checking out some subset $S \subseteq P$. Someone asks you, "hey, give me an *upper bound* of S." You just gotta find *some* $b \in P$ (the larger/global set) that is greater than or equal to every element in S. The person then comes back and says "ok no, I need the *supremum* of S." Now you need to find the *smallest* value out of all the possible upper bounds.



  Hopefully it is clear why this is only relevant in real-valued cases where the "edges" aren't well-defined.

[238] A partially ordered set $(P, \leq)$ is a set of elements such that element i is less than or equal to element $i + i$.

Linear Algebra. Feeling like I need a quick recap from my adv. linalg course. In what follows, let V and W be vector spaces over some field F.

Linear Transformation:

> *A function $T : V \mapsto W$ is called a **linear transformation** from V to W if $\forall x, y \in V$ and $\forall c \in F$:*
>
> - $T(x + y) = T(x) + T(y)$.
> - $T(cx) = cT(x)$.

Now suppose that V and W have ordered bases $\beta = \{v_1, \ldots, v_n\}$ and $\gamma = \{w_1, \ldots, w_m\}$, respectively. Then for each basis vector v_j, there exist unique scalars $a_{ij} \in F$ such that

$$T(v_j) = \sum_{i=1}^{m} a_{ij} w_i \tag{1044}$$

Remember that each v_j and w_i are members of a *vector* space (they are not scalars). And also be careful to not associate the representation of any vector space element with its coordinate vector relative to a specific ordered basis, which *itself is a different linear transformation from some $V \mapsto F^n$*. Keep it abstract.

Matrix Representation of a Linear Transformation:

> *We call the $m \times n$ matrix A defined by $A_{ij} = a_{ij}$ the **matrix representation of T** in the ordered bases β and γ and write $A = [T]_\beta^\gamma$. If $V = W$ and $\beta = \gamma$, then $A = [T]_\beta$.*

Given this definition, I think it's wise to interpret matrix representations by the column-vector point of view. Each column vector $[T(v_j)]_\gamma$, read as "the coordinate vector of $T(v_j)$ relative to ordered basis γ," tells you how each basis vector v_j in domain V gets mapped to a [coordinate] vector in output space W [relative to a given ordered basis γ]. For some reason, my brain has always had a hard time with the fact that the matrix row indices i correspond to output space, while the column indices j represent the input space. The easiest way (I think) to help ground this the right way is to remember that $L_A(x) \triangleq Ax$, i.e. the operator view. At the same time, notice how the effect of Ax is to take successive linear combinations over each element of x.

I just realized another reason why the interpretation felt backwards to my brain: when we are taught matrix multiplication, we do the computations Ax in our heads "row by row" along A, taking inner products of the row with x, so I've been taught to think of the rows as the main "units" of the matrix. I'm not sure how to fix this, but notice that the matrix is basically just a blueprint/roadmap/whatever you want to call it for taking coordinate vectors in one basis to coordinate vectors in another basis. It's really important to remember the coefficients of A are intimately tied to the input/output bases.

Invertibility and Isomorphisms. A function is invertible IFF it is both one-to-one and onto (i.e. bijective). Recall that $rank(T)$ is the dimensionality of the range of T, which is the subspace of W that T maps to.

> *Let $T : V \mapsto W$ be a linear transformation, where V and W are finite-dimensional spaces of <u>equal dimension</u>. Then T is invertible IFF $rank(T) = dim(V)$.*

For any invertible functions T and U:

- $(TU)^{-1} = U^{-1}T^{-1}$. One easy way to show this is

$$(TU)U^{-1}T^{-1}(x) = T(UU^{-1})T^{-1}(x) = TT^{-1}(x) = x \tag{1045}$$

- $(T^{-1})^{-1} = T$. The inverse of T is itself invertible.

Questions:

- **Q:** In general, how can one tell if a matrix $\boldsymbol{A}$ has an eigenvalue decomposition? [insert more conceptual matrix-related questions here . . .]
- **Q:** Let $\boldsymbol{A}$ be real-symmetric. What can we say about $\boldsymbol{A}$?
  - Proof that eigendecomposition $A = Q\Lambda Q^T$ exists: Wow this is apparently quite hard to prove according to many online sources. Guess I don't feel so bad now that it wasn't (and still isn't) obvious.
  - Eigendecomposition not unique. This is apparently because two or more eigenvectors may have same eigenvalue.

This is the principal axis theorem: if A symmetric, then orthonorm basis of e-vects exists.

Stuff I Forget:

- Existence of eigenvalues/eigenvectors. Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$.
  - λ is an eigenvalue of A iff it satisfies $\det(\lambda\boldsymbol{I} - \boldsymbol{A}) = 0$. Why? Because it is an equivalent statement as requiring that $(\lambda\boldsymbol{I} - \boldsymbol{A})\boldsymbol{x} = 0$ has a nonzero solution for $\boldsymbol{x}$.
  - The following statements are equivalent:
    - A is diagonalizable.
    - A has n linearly independent eigenvectors.
  - The **eigenspace** of $\boldsymbol{A}$ corresponding to λ is the solution space of the homogeneous system $(\lambda I - A)x = 0$.
  - $\boldsymbol{A}$ has at most n distinct eigenvalues.

Most info here comes from chapter 5 of your "Elementary Linear Algebra" textbook (around pg305)

- Diagonalizability notes from 5.2 of advanced linear alg. book (261). Recall that $\boldsymbol{A}$ is defined to be diagonalizable if and only if there exists and ordered basis β for the space consisting of eigenvectors of $\boldsymbol{A}$.
  - If the standard way of finding eigenvalues leads to k distinct λ_i, then the corresponding set of k eigenvectors v_i are guaranteed to be linearly independent (but might not span the full space).
  - If $\boldsymbol{A}$ has n linearly independent eigenvectors, then $\boldsymbol{A}$ is diagonalizable.
  - The characteristic polynomial of any diagonalizable linear operator splits (can be factored into product of linear factors). The **algebraic multiplicity** of an eigenvalue λ is the largest positive integer k for which $(t - \lambda)^k$ is a factor of $f(t)$.

Recall that a linear operator is a special case of a linear map where the input space is the same as the output space.

Main Tasks in NLP

Table of Contents Local *Written by Brandon McKinzie*

Going to start compiling a list of the main tasks in NLP (alphabetized). Note that NLP-Progress, a site dedicated to this purpose, is a much more detailed. I'm going for short-and-sweet here.

Constituency Parsing.

- **Task**: Generate parse tree of a sentence. Nodes are typically labeled by parts of speech and/or chunks.
  - A constituency parse tree breaks a text into sub-phrases, or constituents. Non-terminals in the tree are types of phrases, the terminals are the words in the sentence.

Coreference Resolution.

- **Task**: clustering mentions in text that refer to the same underlying real world entities.
- **SOTA**: End-to-end Neural Coreference Resolution.

Dependency Parsing.

- **Task**: Given a sentence, generate its dependency tree (DT). A DT is a labeled directed tree whose nodes are individual words, and whose edges are directed arcs labeled with dependency types.
  - A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between "head" words and words which modify those heads.
- **SOTA**: Deep Biaffine Attention for Neural Dependency Parsing.
  Related Tasks:
  - Constituency parsing. See this great wiki explanation of dependency vs constituency.

Information Extraction.

- **Task**: Given a (typically long) portion of raw text, recover information about pre-specified relations, entities, events, etc.

Language Modeling.

- **Task**: Learning the probability distribution over text sequences. Often used for predicting the next word in a sequence, given the K previous words.
- **SOTA**: ELMo.

Machine Translation.

Semantic Parsing.

- **Task**: Translating natural language into a formal meaning representation *on which a machine can act*.

Semantic Role Labeling.

- **Task**: Given a sentence, extract the predicates[239] and their respective arguments.
- **Historical Approaches**.
  - **CCG Semantic Parsing**. Zettlemoyer & Collins 2005, 2007.
  - Seq2seq. Dong & Lapata, 2016.

Sentiment Analysis.

- **Task**: Determining whether a piece of text is positve, negativ, or neutral.
- **SOTA**: Biattentive classification network (BCN) from Learned in Translation: Contextualized Word Vectors (the CoVe paper) with ELMo embeddings.

Summarization.

Textual Entailment.

- **Task**: Given a premise, determine whether a proposed hypothesis is true.
- **SOTA**: Enhanced Sequenctial Inference (ESIM) model from Enhanced LSTM for Natural Language Inference with ELMo embeddings.

[239] The predicate of a sentence mostly corresponds to the main verb and any auxiliaries that accompany the main verb; whereas the arguments of that predicate (e.g. the subject and object noun phrases) are outside the predicate.

Topic Modeling.

Question Answering. Also called **machine reading comprehension**.

- **Task**: Given a paragraph of text, "read" it and then answer questions pertaining to the text.
- **Dataset**: The main benchmark dataset is the *Stanford Question Answering Dataset* (SQuAD). Each sample has the form `({question, paragraph}, answer)`, where the answer is a subsequence found somewhere in the paragraph (i.e. this is an *extractive* task, not abstractive).
- **SOTA**: Improved versions of the Bidirectional Attention Flow (BiDAF) model, with ELMo embeddings.
- **Related tasks**:

Word Sense Disambiguation.

- **Task**: Associating words in context with their most suitable entry in a pre-defined sense inventory.

TODO: This section is pretty old/redundant by now. Should either move or delete.

BLEU Score. For scoring machine-generated translations when we have access to one or more reference translations.

- Unigram precision: Really naive and basically useless version:

$$P = \frac{\textbf{num pred words that also appear somewhere in ref words}}{\textbf{total num pred words}} \tag{1046}$$

  It's important to emphasize how ridiculous this really is. It literally means that we walk along each word in the prediction and ask "is this word somewhere in any of the reference translations?" and if that answer is "yes", we +1 to the numerator. Period.
- Modified unigram precision: actually considering how many times we've mentioned a given word w when incorporating it into the precision calculation. Now, when walking along a sentence, we add to the aforementioned question, "...and have I seen it less than N times already?" where `N = [max(count(sent, w)) for sent in refs]`. This means our numerator can now be at most N for any given word.
- Generalize to n-grams. Below is the formula for Blue score on n-grams only:

$$p_n(\hat{y}) = \frac{\sum_{\text{ngrams}\in\hat{y}} \text{Count}_{clip}(\text{ngram}, \text{refs})}{\sum_{\text{ngrams}\in\hat{y}} \text{Count}(\text{ngram})} \tag{1047}$$

- Combined Blue score.

$$= \text{BP} \cdot \exp\left\{\frac{1}{4}\sum_{n=1}^{4} \log p_n\right\} \tag{1048}$$

$$\text{BP} = \begin{cases} 1 & \text{len(pred)} > \text{len(ref)} \\ \exp\{1 - \text{len(pred)}/\text{len(ref)}\} & \text{otherwise} \end{cases} \tag{1049}$$

  where BP is the **brevity penalty**.

Coreference Resolution. The task of finding all expressions that refer to the same entity in a text. It is an important step for a lot of higher level NLP tasks that involve natural language understanding such as document summarization, question answering, and information extraction.

Conditional Random Field (CRF). A type of discriminative undirected probabilistic graphical model[240]. The CRF defines the probability, $p(\ell \mid s)$, of a given sentence s being described with the [vector] label ℓ as follows:

See this post by Edwin Chen for a great CRF intro.

$$\text{score}(\ell \mid s) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(s, i, \ell_i, \ell_{i-1}) \tag{1050}$$

$$p(\ell \mid s) = \frac{e^{\text{score}(\ell|s)}}{\sum_{\ell'} e^{\text{score}(\ell'|s)}} \tag{1051}$$

A label element, ℓ_i, could be the POS label for the ith word. The sum in the denominator of (2) is over <u>all</u> possible label combinations for s.

where m is the number of features (e.g. different POS labels) and n is the number of words/tokens in sentence s. So, where's the graph part, you may ask? It can be inferred from the above equation. I like to imagine a picture where I have my set of label nodes $\boldsymbol{Y}$ and set of word nodes $\boldsymbol{X}$ for the sentence s. Note that all nodes are modeled as random variables. The built-in assumption of the CRF is that the value of $\boldsymbol{Y}_i$, the label for the ith word in s, is conditionally independent of the other labels $\boldsymbol{Y}_j$ *given* the previous label $\boldsymbol{Y}_{i-1}$.

- Often used for labeling or parsing of sequential data, e.g. POS tagging, NER.

Question/Answer.

- **Q**: How is this different from logistic regression?
  - **A**: Indeed, this would be identical to logistic regression if we viewed all possible label [vector] combinations ℓ as black boxes with no internal structure (i.e. replace their vector representations with integer ids). The difference is that we are exploiting the sequential interdependencies of the structure of ℓ – we are explicitly modeling the labels as *sequential*.
- **Q**: How is this different from HMMs?
  - **A**: Basically, we can build a CRF equivalent to any HMM, but CRFs are able to define more features and they can have arbitrary weights. Let's look into this in more detail (TODO).
- **Q**: How is this different from basically any neural network-based text classifier? They all accept sequences and output a label prediction.
  - **A**: .

[240] From browsing the literature, it appears that *by far* the most important CRF for sequential labels with NLP is the **linear-chain CRF**, so much so that often authors will say CRF when they really mean linear-chain CRF (and henceforth so will I)