Jacob Chmura

Fundamentals

Probability

Distributions

Definition 1. Let $x \in \{0, ..., N\}$ the **binomial distirbution is**

$$Bin(x|N, \mu) = \frac{N!}{(N-k)!k!}\mu^x(1-\mu)^{N-x} \tag{1}$$

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

$$Ber(x|\mu) = \begin{cases} 1-\mu & if x = 0 \\ \mu & if x = 1 \end{cases} \tag{2}$$

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

Definition 2. If $x \in \{1, .., K\}$ the **categorical** distribution is:

$$Cat(x|\boldsymbol{\theta}) = \prod_{k=1}^{K} \theta_k^{\mathbb{I}(x=k)} \tag{3}$$

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

$$\mathcal{M}(\mathbf{x}|N, \boldsymbol{\theta}) = \frac{N!}{x_1!....x_m!}\prod_{k=1}^{K}\theta_k^{x_k} \tag{4}$$

Definition 3. If $X \in \{0, 1, 2, ...\}$ we that **X is poisson distributed with parameter** $\lambda > 0$ if its probability mass function is:

$$Poi(x|\lambda) = e^{-\lambda}\frac{\lambda^x}{x!} \tag{5}$$

where λ is the mean and variance of x.

Definition 4. The **negative binomial distribution** models the number of successful trails required to observe r failures given a probability of success p and is given by:

$$NegBin(x|r, p) = \frac{x + r - 1!}{x!(r-1!)}(1-p)^r p^x \tag{6}$$

which we can analytically continue to real r using the gamma function Γ.

The moments are: $\mathbb{E}[x] = \frac{pr}{1-p}, \mathbb{V}[x] = \frac{pr}{(1-p)^2}$.

The poisson is a special case of the negative binomial with $Poi(\lambda) = lim_{r\to\infty} NegBin(r, \frac{\lambda}{1+\lambda})$.

When $r = 1$ we get the **geometric distribution**.

Definition 5. The **Gaussian Distribution** is given by:

$$\mathcal{N}(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7}$$

where μ is the mean (and mode) and σ^2 is the variance. Alternatively, $\lambda = \frac{1}{\sigma^2}$ is the **precision**.

Definition 6. The **student t-distribution** is given by:

$$\mathcal{T}_\rho(x|\mu,\sigma^2) = \frac{1}{Z}[1+\frac{1}{\rho}(\frac{x-\mu}{\sigma})^2]^{-\frac{\rho+1}{2}} \tag{8}$$

where μ is the mean, $\sigma > 0$ is the scale parameter and $\rho > 0$ is the **degrees of normality**. Larger values of ρ make the distribution act gaussian.

The gaussian is sensitive to outliers since the probability decays exponentially fast with the squared distance from the center, the student distribution is more robust.

The student distribution is however not log-concave for any parameter value.

If $\rho = 1$ we get the **Cauchy** distribution .

Definition 7. The **laplace distribution** has the pdf:

$$Laplace(x|\mu,b) = \frac{1}{2b}exp(\frac{-|x-\mu}{b}) \tag{9}$$

where μ is a location parameter and $b > 0$ the scale parameter.

The laplace distribution has heavy tails.

Definition 8. We can define **super-gaussian** or **sub-gaussian** distribution based on their **kurtosis**:

$$kurt(z) = \frac{\mu_4}{\sigma^4} = \frac{\mathbb{E}[(Z-\mu)^4]}{(\mathbb{E}[(Z-\mu)^2])^2} \tag{10}$$

where σ is the standard deviation and μ_4 is the 4th **central moment**. The ratio measures how slow the density dies off to zero away from the mean.

- The kurtosis of the gaussian is 3
- A **super-gaussian** (ex. Laplace) has positive excess kurtosis and hence heavier tails
- A **sub-gaussian** (ex. Uniform) has negative excess curtosis and hence lighter tails.

Definition 9. The **gamma distribution** is a flexible distribution for positive real values defined by shape $a > 0$ and rate $b > 0$:

$$Ga(x|a,b) = \frac{b^a}{\Gamma(a)}x^{a-1}e^{-xb} \tag{11}$$

if $a = 1$ we get the **exponential distribution** which describes the times between events in a poisson process.

The **Chi-squared distribution** is a special case where $\chi^2_\rho(x) = Ga(x|\frac{\rho}{2},\frac{1}{2})$. This is the distribution of the sum of ρ squared Gaussian variables.

Definition 10. The **inverse gamma** if the distribution $Y \sim \frac{1}{X}$ where $X \sim Ga(a,b)$. The mean only exists if $a > 1$ and the variance only if $b > 2$.

Definition 11. The **pareto distribution** has the pdf:

$$Pareto(x|m,k) = km^k\frac{1}{x^{k+1}}\mathbb{I}(x \geq m) \tag{12}$$

Definition 12. The **beta distribution** has support over $[0,1]$ and is defined by:

$$Beta(x|a,b) = \frac{1}{B(a,b)}x^{a-1}(1-x)^{b-1} \tag{13}$$

requiring $a, b > 0$.

We can generalize to the **multivariate Dirichlet distribution** which has support over the **probability K-simplex**:

$$Dir(\mathbf{x},\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{k=1}^{K}x_k^{\alpha_k-1}\mathbb{I}(\mathbf{x} \in S_K) \tag{14}$$

Definition 13. The **multivariate guassian** is defined as:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})) \tag{15}$$

where $\boldsymbol{\mu} \in \mathbb{R}^D$ is the mean vector and $\mathbf{\Sigma} \in \mathbb{R}^{D\times D}$ is the covariance matrix.

The term $d_{\mathbf{\Sigma}}(\mathbf{x}, \boldsymbol{\mu})^2 = (\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})$ is the sqaured **Mahalanobis distance** between the data vector $\mathbf{x}$ and the mean vector $\boldsymbol{\mu}$.

- A **full covariance** matrix has $D(D+1)/2$ parameters (symmetric divide by 2)
- A **diagonal covariance** has D parameters on the diagonal
- An **isotopic covariance** has the form $\mathbf{\Sigma} = \sigma^2 I$ and hence one free parameter.
- If $\mathbf{x_n} \sim \mathcal{N}(0, \mathbf{\Sigma})$ then the **scatter matrix** $\mathbf{S} = \sum_n = 1^N \mathbf{x_n}\mathbf{x_n}^T$ has a **Wishart distribution**, $\mathbf{S} \sim Wi(\mathbf{\Sigma}, N)$ where:

  $$Wi(\mathbf{\Sigma}|\mathbf{S}, \rho) = \frac{1}{Z}|\mathbf{\Sigma}|^{\frac{\rho-D-1}{2}} exp(-\frac{1}{2}tr(\mathbf{\Sigma}\mathbf{S^{-1}}) \tag{16}$$

  is defined over positive definite matrices.

Gaussian Soap Bubbles

In high dimensions, the typical set of a gaussian is a thin shell or annulus with distance from origin given by $r = \sigma\sqrt{D}$ and thickness $\mathcal{O}(\sigma D^{\frac{1}{4}}$. This is because the density decays from the origin exponentially, but the volume of a sphere grows exponentially in dimension, and mass is density times volume. The annulus sits where the terms balance.

Marginals and Conditions of MVN

- The marginal of a gaussian is a gaussian. The parameters are found by partitions into block form
- The conditionals of a gaussian are gaussian.
  - The posterior mean of a conditional is linear in the conditioned gaussian
  - The posterior covariance is independent of the conditioned gaussian

Natural Parameters

We can reparamterize via the **precision matrix**: $\mathbf{\Lambda} = \mathbf{\Sigma}^{-1}, \boldsymbol{\eta} = \mathbf{\Sigma}^{-1}\boldsymbol{\mu}$.

Thus the **information form** looks like:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\eta}, \mathbf{\Lambda}) = cexp(\mathbf{x}^T\boldsymbol{\eta} - \frac{1}{2}\mathbf{x}^T\mathbf{\Lambda}x) \tag{17}$$

We can change coordinates to **moment parameters** again via $\mu \to \mathbf{\Lambda}^{-1}\boldsymbol{\eta}, \mathbf{\Sigma} \to \mathbf{\Lambda}^{-1}$.

Linear Gaussian Systems

Definition 14. A **linear gaussian system** is given by $\mathbf{y} \in \mathbb{R}^D, \mathbf{z} \in \mathbb{R}^l$ which are jointly gaussian such that:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu_z}, \mathbf{\Sigma}_z) \tag{18}$$

$$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{y}|\mathbf{W}\mathbf{z} + \mathbf{b}, \mathbf{\Omega}) \tag{19}$$

Then the joint $p(\mathbf{z}, \mathbf{y}) = p(\mathbf{z})p(\mathbf{y}|\mathbf{z})$ is itself a $D + L$ dimensional Gaussian, parameters found by moment matching.

Definition 15. Bayes rule for gaussian linear systems computes $p(\mathbf{z}|\mathbf{y})$, we find the the posterior is itself gaussian given by:

$$p(\mathbf{z}|\mathbf{y}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{20}$$

$$\boldsymbol{\mu} = \boldsymbol{\mu_z} + \boldsymbol{\Sigma_z W}^T(\boldsymbol{\Omega} + \boldsymbol{W\Omega W}^T)^{-1}(\boldsymbol{y} - \boldsymbol{W\mu_z} + \boldsymbol{b}) \tag{21}$$

$$\boldsymbol{\Sigma} = \boldsymbol{\Sigma_z} - \boldsymbol{\Sigma_z W}^T(\boldsymbol{\Omega} + \boldsymbol{W\Omega W}^T)^{-1}\boldsymbol{W\Sigma_z} \tag{22}$$

so if the prior and likelihood is gaussian then the posterior is also gaussian, making the gaussian prior a **conjugate prior** for the gaussian likelihood.

Exponential Families

Definition 16. The **exponential family** is a family of distributions parameterized by $\boldsymbol{\eta} \in \mathbb{R}^K$ with fixed support over $\mathcal{X}^D \subset \mathbb{R}^D$. We say that the distribution is $p(\mathbf{x}|\boldsymbol{\eta})$ in the **exponential family** if its density can be written in the form:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})exp[\boldsymbol{\eta}^T\mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})] \tag{23}$$

where:

- $h(\mathbf{x})$ is the **base measure**
- $\mathcal{T}(\mathbf{x}) \in \mathbb{R}^K$ are the **sufficient statistics**
- $\boldsymbol{\eta}$ are the **natural parameters**
- $A(\boldsymbol{\eta}) = logZ(\boldsymbol{\eta})$ is the **log-partition function** which is convex over $\Omega = \{\boldsymbol{\eta} \in \mathbb{R}^K : A(\boldsymbol{\eta} < \infty)\}$

Typically the natural parameters are taken to be independent of each other, and in this case we say the family is **minimal** and it holds that there is no $\boldsymbol{\eta} \in \mathbb{R}^K - \{0\}$ s.t. $\boldsymbol{\eta}^T\mathcal{T}(\mathbf{x}) = 0$.

If $\boldsymbol{\eta} = f(\phi)$ for some possibly smaller set of parameters ϕ, and the mapping $\phi \mapsto \boldsymbol{\eta}$ is nonlinear, we call this a **curved exponential family**. If f is the identity we say the model is in **canonical form**. If $\mathcal{T}(\mathbf{x}) = \mathbf{x}$ we say this is a **natural exponential family**.

We define the **moment parameters** as the mean of the sufficient statistic vector: $\mathbf{m} = \mathbb{E}[\mathcal{T}(\mathbf{x})]$.

We like exponential families because:

- It is the unique family of distributions with maximal entropy subject to constraints
- It is at the core of gaussian linear systems and variational inference
- It is the only family of distributions with finite size sufficient statistics
- All members of the exponential family have a conjugate prior makign Bayesian inference easier

Example

- Bernoulli, Categorical, Gaussian

Log Partition function is cumulant generating function

Definition 17. The **first and second moments** of a R.V. are $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$

The **first and second cumulatns** of a R.V. are $\mathbb{E}[X]$ and $\mathbb{V}[X]$

Lemma 18. *The derivatives of the log partition function can be used to generate all the cumulants of the sufficient statistics.*

For example:

$$\nabla_\eta A(\boldsymbol{\eta}) = \mathbb{E}[\mathcal{T}(x)] \tag{24}$$
$$\nabla^2_\eta A(\boldsymbol{\eta}) = Cov[\mathcal{T}(x)] \tag{25}$$
$$\tag{26}$$

Natural vs Moment Parameters

Let $\Omega = \{\boldsymbol{\eta} \in \mathbb{R}^K : Z(\boldsymbol{\eta}) < \infty\}$ be the set of normalizable natural parameters.

Definition 19. We say that an exponential family is **regular** if Ω is an open set. Ω is a convex set and $A(\boldsymbol{\eta})$ is a convex function on Ω.

We know that $\mathbf{m} = \nabla_\eta A(\boldsymbol{\eta}) = \mathbb{E}[\mathcal{T}(x)]$ and we define $\mathcal{M} = \{\mathbf{m} \in \mathbb{R}^K : \mathbb{E}_p[\mathcal{T}(\mathbf{x}) = \mathbf{m}]\}$ for some distribution p, to be the set of valid moment parameters.

If the family if minimal, we can moreover map $\boldsymbol{\eta} = \nabla_m A^*(\mathbf{m})$ where

$$A^*(\mathbf{m}) = sup_{\eta\in\Omega} \boldsymbol{\mu}^T \boldsymbol{\eta} - A(\boldsymbol{\eta}) \tag{27}$$

is the convex conjugate of A.

Thus the pair of operators $(\nabla A, \nabla A^*)$ provide coordinate changes between normalizable natural parameters and valid mean parameters.

MLE For the Exponential Family

The likelihood of an exponential family has the form:

$$p(\mathcal{D}|\boldsymbol{\eta}) \propto exp[\boldsymbol{\eta}^T \mathcal{T}(\mathcal{D}) - N \cdot A(\boldsymbol{\eta})] \tag{28}$$

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

$$\mathcal{T}(\mathcal{D}) = [\sum_{n=1}^{N} \mathcal{T}_1(\mathbf{x_n}), ..., \sum_{n=1}^{N} \mathcal{T}_K(\mathbf{x_n})] \tag{29}$$

Theorem 0.20 (**Pitman-Koopman-Darmois**)**.** *The exponential family is the only family of distribution with finite sufficient statistics*

$$p(\mathcal{D}|\boldsymbol{\eta}) = p(\mathcal{T}(\mathcal{D})|\boldsymbol{\eta}) \tag{30}$$

Hence, we can perform **moment matching** to find that at the MLE:

$$\mathbb{E}[\mathcal{T}(\mathbf{x})] = \frac{1}{N_D} \sum_{n=1}^{N} \mathcal{T}(\mathbf{x}_n) \tag{31}$$

Maximum Entropy Family

Theorem 0.21. *Suppose we want to find a distribution $p(\boldsymbol{x})$ to describe data, where we know the expected values F_k of certian features or function $f_k(\boldsymbol{x})$:*

$$\int f_k(\boldsymbol{x}) p(\boldsymbol{x}) d\boldsymbol{x} = F_k \tag{32}$$

and we have a prior $q(\boldsymbol{x})$.

We search for the distribution that is as close to our prior $q(\boldsymbol{x})$ in the sense of KL divergence. Setting $q(\boldsymbol{x}) \propto 1$ results in the ***maximum entorpy model****.*

Then p is an exponential family of the form:

$$p(\boldsymbol{x}|\boldsymbol{\lambda}) = \frac{q(\boldsymbol{x})}{Z} exp(-\sum_k \lambda_k f_k(\boldsymbol{x})) \tag{33}$$

Fisher Information Matrix

Definition 22. The **score function** is defined to be the gradient of the log likelihood

$$\mathbf{s}(\boldsymbol{\theta}) = \nabla logp(\mathbf{x}|\boldsymbol{\theta}) \tag{34}$$

Lemma 23. *The expected value of the score function is zero*

$$\mathbb{E}_{p(x|\theta)}[\nabla logp(\boldsymbol{x}|\boldsymbol{\theta}) = 0] \tag{35}$$

Definition 24. The **Fisher information matrix** is defined to be the covariance of the score function:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{x\sim p(x|\theta)}[\nabla logp(\mathbf{x}|\boldsymbol{\theta}) \nabla logp(\mathbf{x}|\boldsymbol{\theta})^T] \tag{36}$$

Theorem 0.25. *The Fisher information matrix equals the expected hessian of the negative log likelihood:*

$$\boldsymbol{F}(\boldsymbol{\theta})_{ij} = -\mathbb{E}_{\mathbf{x}\sim\theta}[\frac{\partial^2}{\partial\theta_i\theta_j} logp(\boldsymbol{x}|\boldsymbol{\theta})] \tag{37}$$

Approximating KL divergence with Fisher Information Matrix

Lemma 26. *Mahalanobis distance based on the fisher information can be viewed as an approximation to the kl divergence between two distributions*

Let $p_\theta(\boldsymbol{x}), p_{\theta'}(\boldsymbol{x})$ be two distributions with parameters like $\boldsymbol{\theta'} = \boldsymbol{\theta} + \boldsymbol{\delta}$ in euclidean space.

Instead of comparing in euclidean parameter space, we can measure how close the second distribution is to the first in terms of their predictive distirbution:

$$D_{KL}(p_\theta||p_{\theta'}) = \mathbb{E}_{p_\theta(x)}[logp_\theta(\boldsymbol{x}) - logp_{\theta'}(\boldsymbol{x})] \tag{38}$$

A second order taylor series approximation of the KL yields:

$$D_{KL}(p_\theta||p_{\theta'}) \approx -\boldsymbol{\delta}^T E[\nabla logp_\theta(\boldsymbol{x})] - \frac{1}{2}\boldsymbol{\delta}^T \mathbb{E}[\nabla^2 logp_\theta(\boldsymbol{x})]\boldsymbol{\delta} \tag{39}$$

where the first term vanishes since the expected score function is zero.

Thus:

$$D_{KL}(p_\theta||p_{\theta'}) \approx \frac{1}{2}\boldsymbol{\delta}^T \boldsymbol{F}(\boldsymbol{\theta})\boldsymbol{\delta} \tag{40}$$

Hence ***the fisher information yields an approximate coordinate change from parameter space distances to distributional distance measures by KL****.*

Fisher Information Exponential Families

$$\mathbf{F}_\eta = Cov[\mathcal{T}(\mathbf{x})] \tag{41}$$

$$\mathbf{F}_m = Cov[\mathcal{T}(\mathbf{x})]^{-1} \tag{42}$$

Transformation of random variables

Suppose $\mathbf{x} \sim p_x(\mathbf{x})$ and $\mathbf{y} = f(\mathbf{x})$ for a deterministic function. Goal is to compute $p_y(\mathbf{y})$.

Bijections

If f is a bijection, the **change of variables** formula produces:

$$p_y(\mathbf{y}) = p_x(f^{-1}(\mathbf{y}))|det[\mathbf{J}_{f^{-1}(\mathbf{y})}]| \tag{43}$$

where $\mathbf{J}_{f^{-1}}$ is the Jacobian of the inverse mapping f^{-1} evaluated at $\mathbf{y}$.

Monte Carlo Approximation

If we cannot compute the jacobian, we can approximate by drawing S samples $\mathbf{x}^s \sim p(\mathbf{x})$, computing $\mathbf{y}^s = f(\mathbf{x})^s$ and the construction the empirical pdf:

$$p_{\mathcal{D}}(\mathbf{y}) = \frac{1}{S}\sum_{s=1}^{S} \delta(\mathbf{y} - \mathbf{y}^s) \tag{44}$$

Note: the mode of the transformed distribution is not the transform of the original mode in general

Probability Integral Transform

Suppose X is a R.V. with cdf P_X, and let $Y(X) = P_X(X)$ be a transformation of X.

Lemma 27. *Y has a uniform distribution:*

$$P_Y(y) = Pr(Y \leq y) = Pr(P_X(X) \leq y) = Pr(X \leq P_X^{-1}(y)) = P_X(P_X^{-1}(y)) = y \tag{45}$$

Lemma 28. *Kolmogorov-Smirnov test to see if a set of samples come from a given distribution.*

Sample $x_n \sim p_x$, compute the empirical cdf of $Y = P_X(X)$ by computing $y_n = P_X(x_n)$ and then sorting.

Then we compute the theoretical cdf of the distribution and compute the KS statistic which is the maximum distance between these two curves:

$$D_n = max_x|P_n(x) - P(x)| \tag{46}$$

Markov Chains

Definition 29. Markov Assumption:

$$p(\mathbf{x}_{t+\tau}|\mathbf{x}_t, \mathbf{x}_{1:t-1}) = p(\mathbf{x}_{t+\tau}|\mathbf{x}_t) \tag{47}$$

Then we can write the joint distribution for finite length sequence as:

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_1)\prod_{t=2}^{T} p(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{48}$$

Markov Transition Kernels

Definition 30. The condition $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is called the **Markov kernel**.

The model is **stationary** if the transition function $p(\mathbf{x}_t|x_{1:t-1})$ is independent of time.

In the case of discrete variables, we can encode the $K - ary$ state transitions in a **transition matrix** $A \in \mathbb{R}^{K\times K}$. where A_{ij} = the probability of going from state i to state j

The **n-step transition matrix A(n)** is defined as:

$$A_{ij}(n) = p(X_{t+n} = j|X_t = i) \tag{49}$$

which is the probability of getting from i to j in exactly n steps.

Lemma 31. *Chapman-Kolmogorov equations state:*

$$A_{ij}(m+n) = \sum_{k=1}^{K} A_{ik}(m) A_{kj}(n) \tag{50}$$

Definition 32. A **n-th order markov assumption** stores a memory of length n:

$$p(\mathbf{x}_{1:T}) = p(\mathbf{x}_{1:n}) \prod_{t=n+1}^{T} p(\mathbf{x}_t | \mathbf{x}_{t-n:t-1}) \tag{51}$$

Note: can always convert a higher order model to a first order one by defining an augmented state space that contains the past n observations.

Parameter Estimation

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

$$p(x_1 : t|\boldsymbol{\theta}) = \pi(x_1) A(x_1, x_2) \cdot ... \cdot A(x_{T-1}, x_T) \tag{52}$$

and hence the log-likelihood of a set of sequences $\mathcal{D} = (\mathbf{x_1}, ..., \mathbf{x_N})$ where $\mathbf{x_i} = (x_{i1} ..., x_{iT_i})$ is:

$$logp(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N} logp(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_{j} N_j^1 log\pi_j + \sum_{j,k} N_{jk} log A_{jk} \tag{53}$$

where:

$$N_j^1 = \sum_{i=1}^{N} \mathbb{I}[x_{i1} = j] \tag{54}$$

$$N_{jk} = \sum_{i=1}^{N} \sum_{t=1}^{T_i - 1} \mathbb{I}[x_{it=j, x_{i,(t+1)=k}}] \tag{55}$$

$$N_j = \sum_{j} N_{jk} \tag{56}$$

and by the method of **lagrange multipliers**

$$\hat{\pi}_j = \frac{N_j^1}{\sum_{j'} N_{j'}^1} \tag{57}$$

$$\hat{A_{jk}} = \frac{N_{jk}}{N_j} \tag{58}$$

Fitting n-gram models will overfit due to data sparsity, we will see exponentially less contexts for larger memory lengths as state space grows leading to zero counts. Map estimation places a uniform dirichlet prior to overcome.

Stationary Distribution and Ergodicity

Definition 33. Let A be the one step transition matrix and $\pi_t(j)$ be the probability of being in state j at time t.

Given initial distribution over states π_o then :

$$\pi_1(j) = \sum_{i} \pi_o(i) A_[ij] \tag{59}$$

and hence:

$$\boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{A} \tag{60}$$

If we iterate the system and reach a state where $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A}$

then $\boldsymbol{\pi}$ is the **stationary distribution of the markov chain**.

To find the stationary distribution we solve the eigenvector equation $\mathbf{A}^T\mathbf{v} = \mathbf{v}$ and set $\boldsymbol{\pi} = \mathbf{v}^T$, ensuring to normalize to norm 1.

- The eigenvectors are only guaranteed to be real-values if all entries in the matrix are strictly positive

A necessary condition to have a unique stationary distribution is that the state transition diagram be a singly connected component. Such chains are called **irreducible**.

Definition 34. A chain has a **limiting distribution** if $\pi_j = lim_{n\to\infty} A_{ij}^n$ exists and independent of the starting state i, for all j.

A chain is **regular** if $A_{ij}^n > 0$ for some integer n and all i, j, meaning it is possible to get from any state to any other state after some n steps.

A state i is **aperiodic** if the period $d(i) := gcd\{t : A_{ii}(t) > 0\} = 1$

Theorem 0.35. *Every irreducible (singly connected) aperiodic finite state Markov chain has a limiting distribution which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

Definition 36. For infinite state spaces, we say a chain is **reccurent** if you will return to every state with probabilitity 1. We say it is **non-null recurrent** if the expected time to return is finite for all states.

A chain is **ergodic** if it is aperiodic, recurrent and non-null.

Theorem 0.37. *Every irreducible, ergodic Markov chain has a limiting distribution which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

Establishing ergodicity is dificult. An easier condition to verify is to check the *detailed balance equations.*

Definition 38. A markov chain is **time reversible** if there exists a distribution $\boldsymbol{\pi}$ s.t.:

$$\pi_i A_{ij} = \pi_j A_{ji} \tag{61}$$

the **detailed balance equations are satisfied.**

Theorem 0.39. *If a markov chain with transition matrix $\boldsymbol{A}$ is regular and satisfies the detailed balance equations w.r.t $\boldsymbol{\pi}$ then $\boldsymbol{\pi}$ is a stationary distribution of the chain.*

Divergences in the space of distributions

Given two distribution P, Q defined on the same measure space (X, M, μ).

f-divergence

Definition 40. Let $f : \mathbb{R}^+ \to \mathbb{R}$ be convex function with f(1) = 0.

We define the parametric divergence:

$$D_f(p||q) = \int q(\mathbf{x}) f(\frac{p(\mathbf{x})}{q(\mathbf{x}}) d\mathbf{x} \tag{62}$$

which is non-negative by *Jensens inequality.*

If we take $f(r) = rlogr$ then we get the **Kullback Leibler Divergence**:

$$D_{KL}(p||q) = \int p(\mathbf{x}) log \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} = \mathbb{E}_{x\sim p}[log \frac{p(\mathbf{x})}{q(\mathbf{x})}] \tag{63}$$

If we take $f(x) = \frac{4}{1-\alpha^2}(1 - x^{\frac{1+\alpha}{2}})$ the f-divergence becomes the **alpha divergence**:

$$D_\alpha^A(p||q) = \frac{4}{1-\alpha^2}(1 - \int p(\mathbf{x})^{\frac{1+\alpha}{2}} q(\mathbf{x})^{\frac{1-\alpha}{2}} d\mathbf{x}) \tag{64}$$

where $\alpha \neq \pm 1$.

Lemma 41.

$$lim_{\alpha\to 0} D_\alpha^A = D_{KL}(q||p) \tag{65}$$
$$lim_{\alpha\to 1} D_\alpha^A = D_{KL}(p||q) \tag{66}$$

Integral Probability Metrics

Definition 42. Let $\mathcal{F}$ be a class of smooth functions.

The **integral probability metric** is defined as:

$$D_{\mathcal{F}}(P, Q) = sup_{f\in\mathcal{F}} |\mathbb{E}_p[f(\mathbf{x})] - \mathbb{E}_q[f(x)]| \tag{67}$$

Suppose $\mathcal{F}$ is the set of functions with bounded Lipschitz constant:

$$\mathcal{F} = \{||f||_L \leq 1\} \tag{68}$$
$$||f||_L = sup_{x\neq x'} \frac{|f(\mathbf{x}) - f(\mathbf{x'})|}{||\mathbf{x} - \mathbf{x'}||} \tag{69}$$

In this case, the integral probability metric produces the **Wasserstein-1 distance**:

$$W_1(P, Q) = sup_{||f||_L\leq 1} |\mathbb{E}_p[f(\mathbf{x})] - \mathbb{E}_q[f(\mathbf{x})]| \tag{70}$$

Suppose instead $\mathcal{F}$ is a **reproducing kernel hilbert space**, defined by a positive definite kernel $\mathcal{K}$.

Then we can represent functions in this class as an infinite sum of basis functions for a hilbert space basis:

$$f(\mathbf{x}) = \langle f, \phi(\mathbf{x})\rangle_{\mathcal{F}} = \sum_l f_l \phi_l(\mathbf{x}) \tag{71}$$

If we restrict to witness function in the hilbert space unit ball: $||f||_{\mathcal{F}}^2 = \sum_l f_l^2 \leq 1$, the linearity of expectations implies:

$$\mathbb{E}_p[f(\mathbf{x})] = \langle f, \mathbb{E}_p[\phi(\mathbf{x})]\rangle_{\mathcal{F}} = \langle f, \boldsymbol{\mu}_P\rangle_{\mathcal{F}} \tag{72}$$

where $\boldsymbol{\mu_P}$ is the **kernel mean embedding of distribution P**.

In this case, the integral probability metric produces the **Maximum mean discrepancy**:

$$MMD(P, Q, \mathcal{F}) = sup_{||f||\leq 1} \langle f, \boldsymbol{\mu}_P - \boldsymbol{\mu}_Q\rangle_{\mathcal{F}} = \frac{\boldsymbol{\mu}_P - \boldsymbol{\mu_Q}}{||\boldsymbol{\mu}_P - \boldsymbol{\mu}_Q||} \tag{73}$$

Computing the MMD with the Kernel Trick

We are given $\mathcal{X} = \{\mathbf{x}_n\}_{n=1}^N$ and $\mathcal{X}' = \{\mathbf{x}'_m\}_{m=1}^M$ where $\mathbf{x}_n \sim P$ and $\mathbf{x}'_m \sim Q$.

We construct the empirical kernel mean embeddings:

$$\boldsymbol{\mu}_P = \frac{1}{N}\sum_{n=1}^{N}\phi(\mathbf{x}_n) \tag{74}$$

$$\boldsymbol{\mu}_Q = \frac{1}{M}\sum_{m=1}^{M}\phi(\mathbf{x}'_m) \tag{75}$$

Then expanding the squared MMD only involves inner products of feature vectors and can be computed as:

$$MMD^2(\mathcal{X},\mathcal{X}') = \frac{1}{N^2}\sum_{n=1}^{N}\sum_{n'=1}^{N}\mathcal{K}(\mathbf{x}_n,\mathbf{x}_{n'}) - \frac{2}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}\mathcal{K}(\mathbf{x}_n,\mathbf{x'}_m) + \frac{1}{M^2}\sum_{m=1}^{M}\sum_{m'=1}^{M}\mathcal{K}(\mathbf{x'}_m,\mathbf{x'}_{m'}) \tag{76}$$

which takes $\mathcal{O}(n^2)$ time to compute where N is the number of samples from each distribution.

For high dimensional data such as images, we often pre train a CNN and define our kernel in terms of hidden features from the CNN. This is caleld ***kernel inception distance****.*

Total Variational Distance

Definition 43. The **total variational distance** between probability distributions is defined as:

$$D_{TV}(p,q) = \frac{1}{2}\int |p(\mathbf{x}) - q(\mathbf{x})| d\mathbf{x} \tag{77}$$

which is the only f-divergence which is also an integral probability measure.

Density Ratio Estimation using Binary Classifiers

Definition 44. Label points from P with $y = 1$ and points from Q with $y = 0$.

Let $p(y = 1) = \pi$ be the class prior and use Bayes rule to get the density ratio $r(\mathbf{x}) = \frac{P(\mathbf{x})}{Q(\mathbf{x})}$:

$$\frac{P(\mathbf{x})}{Q(\mathbf{x})} = \frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=0)} \tag{78}$$

$$= \frac{p(y=1|\mathbf{x})p(\mathbf{x})}{p(y=1)} / \frac{p(y=0|\mathbf{x})p(\mathbf{x})}{p(y=0)} \tag{79}$$

$$= \frac{p(y=1|\mathbf{x})}{p(y=0|\mathbf{x})}\frac{1-\pi}{\pi} \tag{80}$$

The **density ratio estimation** involves assuming $\pi = 0.5$, then fitting a binary classifier $h(\mathbf{x}) = p(y = 1|\mathbf{x})$ and computing $r = \frac{h}{1-h}$

Theorem 0.45. *Every f-divergence can be recovered by fitting the density ratio estimaiton with a binary classifier for some loss function* l*.*

Ex: total variational distance corresponds to hinge loss: $l(y, h) = max(0, 1 - yh)$

Theorem 0.46. *A trained discriminator plays the role of the witness function in an equivalent integral probability measure under a parameterized class of functions.*

Bayesian Statistics

Probability theory is concerned with the forward mapping of parameters to data $(\boldsymbol{\theta}, \mathbf{x}) \mapsto \mathbf{y}$.

Statistics is concerned with the inverse problem of infering unknown parameters $\boldsymbol{\theta}$ *given observation* $\boldsymbol{x}$

Frequentist vs. Bayesian Statistics

Frequentist statistics is based on the concept of *repeated trails*. We are given an **estimator** such as the maximum likelihood estimator $\hat{\boldsymbol{\theta}}(\mathcal{D}) = argmax_\theta p(\mathcal{D}|\boldsymbol{\theta})$.

Definition 47. The **sampling distribution of an estimator** involves applying the estimator to multiple random datasets of the same size, drawn from the same, but known *true distribution* $p^*(\mathcal{D})$, and looking at the distribution of estimated values:

$$\{\hat{\boldsymbol{\theta}}(\mathcal{D}') : \mathcal{D}' \sim p^*\} \tag{81}$$

Bayesian statistics treats the known parameters $\boldsymbol{\theta}$ as a random variable, and hence model them with a probability distribution. *We consider the data fixed*, and are interested in the distribution:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})}{p(\mathcal{D})} = \frac{p(\boldsymbol{\theta})p(\mathcal{D}|\boldsymbol{\theta})}{\int p(\mathcal{D}, \boldsymbol{\theta}')p(\boldsymbol{\theta}')d\boldsymbol{\theta}'} \tag{82}$$

- $p(\boldsymbol{\theta})$ is the **prior distribution** encoding our initial beliefs
- $p(\mathcal{D}|\boldsymbol{\theta})$ is the **likelihood function** which encodes how the data depends on the parameters
- $p(\boldsymbol{\theta}|\mathcal{D})$ is the **posterior distribution we want**
- $p(\mathcal{D})$ is the **evidence** and is computationally difficult due to the high dimensional integral

The task of computing the posterior is called **baysian inference**.

Why Bayesian

- The theorem tells us if our data is exchangeable then there **must exist** a parameter $\boldsymbol{\theta}$ under which the Bayesian approach follows immediately due to the existance of a likelihood $p(\mathbf{x}_i|\boldsymbol{\theta})$ and a prior $p(\boldsymbol{\theta})$

  Theorem 0.48 (De Finetti)**.** *A sequence of random variables* $(\mathbf{x_1}, ...)$ *is* ***infinitely exchangeable*** *if for any n, the joint* $p(\mathbf{x}_1, ..., \mathbf{x}_n)$ *is invariant to permutation of the indices.* Note: IID implies exchangeability but not vice versa

  A sequence of random variables is infinitely exchangeable iff for all n:

$$p(\boldsymbol{x}_1, .., .\boldsymbol{x}_n) = \int \prod_{i=1}^{n} p(\boldsymbol{x}_i, \boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \tag{83}$$

  where $\boldsymbol{\theta}$ *is some hidden (possibly infinite dimensional) random variable.*

  Hence x_i ***are idd conditional on*** $\boldsymbol{\theta}$

- Bayesian approach also allows us to solve problems that don't repeat, which is undefined under frequentest statistics. Ex: find probability of ice caps melting in 2030
- Decision Theory optimality

  Theorem 0.49 (Dutch Book Theorem)**.** *Any method for representing uncertainty that is not bayesian is guaranteed to cause a decision maker to incur regret in the context of decision theory.*

- Bayesian approach lends itself to **online learning**. Once we have computed the posterior we can *throw away the data and whatever prior knowledge we had*, since we can recursively update our beliefs:

$$p(\boldsymbol{\theta}|\mathcal{D}_{1:t}) \propto p(\mathcal{D}_t|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D}_{1:t-1}) \tag{84}$$

  so we can rapidly update our beliefs even with small amounts of new data and changing distributions to perform sequential decision making under uncertainty.

Why not just MAP estimation?

We can avoid computing $p(\mathcal{D})$ and put:

$$\hat{\boldsymbol{\theta}} = argmax_\theta[logp(\mathcal{D}|\boldsymbol{\theta}) + logp(\boldsymbol{\theta})] \tag{85}$$

with some prior.

- The MAP estimate gives no measure of uncertainty. We don't know how much to trust our parameter estimate
- The plugin approximation does not capture predictive uncertainty. In machine learning, often we care about the **predictive uncertainty** since our parameters are **unidentifiable**. Hence we derivative uncertainty in the predictions by computing the **posterior predictive distribution**:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta} \tag{86}$$

  which **integrated out** the unknown parameters and hence reduces the chance of **overfitting** If we approximate the posterior with a point estimate using a **delta function** $p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})$ we can using the sifting property of the delta function to get the **plugin approximation to the posterior predictive distribution**:

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})\delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})d\boldsymbol{\theta} = p(\mathbf{y}|\mathbf{x}, \hat{\boldsymbol{\theta}}) \tag{87}$$

  This plugin approximation *ignores uncertainty in the parameter estimates which can underestimate the uncertainty.* We do model the **aleatoric uncertainty** since it would persist even if we knew the true model and true paramters. In practice we don't even know the parameters, which results in an orthogonal source of uncertainty called **epistemic uncertainty** which arises due to lack of knowledge about the true. This is only taken into account with the bayesian approach.

- The MAP estimate is often untypical of the posterior. The mode is a bad summary statistic
- The MAP estimate is only optimal for the 0-1 loss. An improvement is the **hamming loss** which arises by the vector of **max marginals** and is called **maximizer of posterior marginals MPM estimate**:

$$\hat{\boldsymbol{\theta}} = [argmax_{\theta_d} \int_{\theta_{-d}} p(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta}_{-d}]_{d=1}^D \tag{88}$$

  which depends on the whole distribution

- The MAP estimate is not invariant to reparameterization. MLE doesn't suffer since the likelihood is a function not a density. Bayesian inference does not suffer since the change of measure is taken into account when itnegrating over parameter space.
- MAP estimation cannot handle the cold start problem. Even in big data we are often in small data regime since natural data maye have **long tails** where a few important events occur rarely, and hence have high uncertainty.

Conjugate Priors

Definition 50. A prior $p(\boldsymbol{\theta}) \in \mathcal{F}$ is a **conjugate prior for a likelihood function** $p(\mathcal{D}|\boldsymbol{\theta})$ if the posterior $p(\boldsymbol{\theta}|\mathcal{D}) \in \mathcal{F}$.

Examples:

- The conjugate prior to the binomial likelihood is the beta distribution. We get a beta posterior.
- The conjugate prior to the categorical distribution is the dirichlet distribution. We get a dirichlet posterior.
- The conjugate prior to a gaussian distributon with fixed prevision is a gaussian.
- The conjugate prior to a gaussian distribution is the normal inverse gamma distribution.

Conjugate Priors for the Exponential Family

The likelihood of the exponential family is of the form:

$$p(\mathcal{D}, \boldsymbol{\eta}) = h(\mathcal{D}) exp(\boldsymbol{\eta}^T s(\mathcal{D}) - N_D A(\boldsymbol{\eta})) \tag{89}$$

where $s(\mathcal{D}) = \sum_{n=1}^{N} s(\mathbf{x}_n)$ and $h(\mathcal{D}) = \prod_{n=1}^{N} h(\mathbf{x})_n$.

We write the prior in the form:

$$p(\boldsymbol{\eta}|\boldsymbol{\tau}, \rho) = \frac{1}{Z(\boldsymbol{\tau}, \rho)} exp(\boldsymbol{\tau}^T \boldsymbol{\eta} - \rho A(\boldsymbol{\eta})) \tag{90}$$

where ρ is the strength of the prior, and $\boldsymbol{\tau}/\rho$ is the prior mean, and $Z(\boldsymbol{\tau}, \rho)$ is a normalizing factor.

The posterior distribution is given by:

$$p(\boldsymbol{\eta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\eta}) p(\boldsymbol{\eta})}{p(\mathcal{D})} \tag{91}$$

$$= \frac{h(\mathcal{D})}{Z(\boldsymbol{\tau}, \rho) p(\mathcal{D})} exp((\boldsymbol{\tau} + s(\mathcal{D}))^T \boldsymbol{\eta} - (\rho + N_D) A(\boldsymbol{\eta})) \tag{92}$$

$$= \frac{1}{Z(\boldsymbol{\tau}', \rho')} exp(\boldsymbol{\tau}'^T \boldsymbol{\eta} - \rho' A(\boldsymbol{\eta})) \tag{93}$$

where $\boldsymbol{\tau}' = \boldsymbol{\tau} + s(\mathcal{D})$, $\rho' = \rho + N_D$, $Z(\boldsymbol{\tau}', \rho') = \frac{Z(\boldsymbol{\tau}', \rho')}{h(\mathcal{D})} p(\mathcal{D})$ which is an exponential family.

The posterior mean $\mathbb{E}[\boldsymbol{\eta}|\mathcal{D}] = \frac{\boldsymbol{\tau}'}{\rho'}$ is given by a convex combination of the prior mean and the MLE.

The predictive density for future observables $\mathcal{D}' = (\mathbf{x}'_1, ..., \mathbf{x}'_{N'})$ given past data $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_N)$ is:

$$p(\mathcal{D}'|\mathcal{D}) = \int p(\mathcal{D}'|\boldsymbol{\eta}) p(\boldsymbol{\eta}|\mathcal{D}) d\boldsymbol{\eta} \tag{94}$$

$$= \int h(\mathcal{D}') exp(\boldsymbol{\eta}^T s(\mathcal{D}') - N' A(\boldsymbol{\eta})) \frac{1}{Z(\boldsymbol{\tau}', \rho')} exp(\boldsymbol{\eta}^T \boldsymbol{\tau}' - \rho' A(\boldsymbol{\eta})) d\boldsymbol{\eta} \tag{95}$$

$$= h(\mathcal{D}') \frac{Z(\boldsymbol{\tau}' + s(\mathcal{D}) + s(\mathcal{D}'), \rho + N + N')}{Z(\boldsymbol{\tau}' + s(\mathcal{D}, \rho' + N))} \tag{96}$$

Robust Heavy Tail Priors

The assessment of the influence of the prior on the posterior is called **sensitivity analysis**. Heavy tail priors are a way to create **robust priors**.

Often heavy-tailed priors are not conjugate, but we can often approximate with a (possibly infinite) mixture of conjugate priors.

NonInformative Priors

When we have little domain knowledge it is desirable to use a **noninformative prior**.

- Maximum entropy prior under constraints (will be exponential family)
- Jeffers Pior: aims to achieve invariance under some invertible transformation so that the posterior does not depend on how we paramterize the model. Given by:

$$p(\boldsymbol{\theta}) \propto \sqrt{det\mathbf{F}(\boldsymbol{\theta})} \tag{97}$$

  where $\mathbf{F}$ is the Fisher informaion matrix.
- Translation invariant priors, scale invariant priors
- Learn invariant priors by solving a variational optimization problem
- Reference priors: define a prior as a distribution which is maximally far form all possible posteriors, when averaged over datasets. $p(\boldsymbol{\theta})$ is a reference prior if it maximizes the expected KL divergence between posterior and prior:

$$p^*(\boldsymbol{\theta}) = argmax_{p(\theta)} \int_{\mathcal{D}} p(\mathcal{D}) D_{KL}(p(\boldsymbol{\theta}|\mathcal{D})||p(\boldsymbol{\theta}))d\mathcal{D} \tag{98}$$

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \tag{99}$$

  In 1d, this is equivalent to Jeffreys prior, in higher dimensions, we compute one parameter at a time using chain rule if tractable else use variational inference.

Hierarchical Priors

Definition 51. A bayesian model requires specificying a prior $p(\boldsymbol{\theta})$ on the paramters. The parameters of the prior are called **hyperparameters**, $\boldsymbol{\phi}$

Definition 52. If the hyperparamters $\boldsymbol{\phi}$ are unknown, we can put a prior on them. This defines a **hierarchical bayesian model** like $\boldsymbol{\phi} \rightarrow \boldsymbol{\theta} \rightarrow \mathcal{D}$. so the joint factorizes:

$$p(\boldsymbol{\phi}, \boldsymbol{\theta}, \mathcal{D}) = p(\boldsymbol{\phi})p(\boldsymbol{\theta}|\boldsymbol{\phi})p(\mathcal{D}|\boldsymbol{\theta}) \tag{100}$$

*Use case: Suppose we have $J > 1$ related datasets $\mathcal{D}_j$, each with their own parameters $\boldsymbol{\theta}$. Inferring $p(\boldsymbol{\theta}|\mathcal{D})$ independantly for each group will give poor results if some of the datasets are small. A hierarchical model allows us to borrow **statistical strength** from groups with lots of data, and hence well informed posteriors.*

Empircal Bayes

Posterior inference in hierarchical models can be computationally challenging. A convient approximation is to first compute a point estimate of the hyperparameters $\hat{\boldsymbol{\phi}}$ and then compute the conditional posterior $p(\boldsymbol{\theta}|\hat{\boldsymbol{\phi}}, \mathcal{D})$ rhan than the joint posterior $p(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathcal{D})$.

We can estimate the hyper-paramters by maximizing the marginal likelihood:

$$\hat{\boldsymbol{\phi}}_{mml}(\mathcal{D}) = argmax_{\phi} p(\mathcal{D}|\boldsymbol{\phi}) = argmax_{\phi} \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})|\boldsymbol{\phi})d\boldsymbol{\theta} \tag{101}$$

which is known as **type 2 maximum likelihood** since we are optimizing over the hyperparamters rather than the parameters.

Since we are estimating the prior paramters from data, this approach is **empirical bayes**.

Model Selection and Evaluation

Bayesian Model Selection

Using bayes rule:

$$\hat{m} = argmax_{m \in \mathcal{M}} p(m|\mathcal{D}) = argmax_{m \in \mathcal{M}} \frac{p(\mathcal{D}|m)p(m)}{\sum_m p(\mathcal{D}|m)p(m)} \tag{102}$$

Assuming a uniform prior over the models, $p(m) = \frac{1}{|\mathcal{M}|}$ then the MAP is:

$$\hat{m} = argmax_{m \in \mathcal{M}} p(\mathcal{D}|m) = argmax_{m \in \mathcal{M}} \int p(\mathcal{D}|\boldsymbol{\theta}, m)p(\boldsymbol{\theta}|m)d\boldsymbol{\theta} \tag{103}$$

which is known as the **marginal likelihood for model m**. This needs to be estimates with variational Bayes or sequential monte carlo.

Cross Validation and Marginal Likelihood

Leave one out cross validation optimizes the loss:

$$L_{LOO}(m) = \sum_{n=1}^{N} logp(\mathcal{D}_n|\hat{\boldsymbol{\theta}}(\mathcal{D}_{-n}), m) \tag{104}$$

On the other hand, we can write the log marginal likelihood:

$$LML(m) = logp(\mathcal{D}|m) = log \prod_{n=1}^{N} p(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) = \sum_{n=1}^{N} logp(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) = \sum_{n=1}^{N} logp \int p(\mathcal{D}_n|\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathcal{D}_{1:n-1}, m)d\boldsymbol{\theta} \tag{105}$$

A *plugin approximation to the nth predictive distribution* results in:

$$logp(\mathcal{D}|m) \approx \sum_{n=1}^{N} logp(\mathcal{D}_n)|\hat{\boldsymbol{\theta}}(\mathcal{D}_{1:n-1}), m) \tag{106}$$

Conditional Marginal Likelihood

The marginal likelihood answers *what is the likelihood of generating the training data from my prior*. We are interested in *what is the probability that the posterior could generate withheld points from the data distribution* which is related to generalization of the fitted model. Indeed the LML decomposition may overfit.

Definition 53. The **conditional marginal likelihood** is define:

$$CLML(m) = \sum_{n=K}^{N} logp(\mathcal{D}_n|\mathcal{D}_{1:n-1}, m) \tag{107}$$

where $K \in \{1, ..., N\}$ is a parameter of the algorithm.

This evaluates the LML of the last N-K datapoints under the posterior given by the first K datapoints.

Bayesian leave one out estimate

Goal: efficiently approximate the leave-one-out estimate without have to fit the model N times on conditional supervised models ($p(\mathcal{D}|\boldsymbol{\theta}) = p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$).

Suppose we have computed the posterior given the full dataset for model m. Then we can evaluate the predictive distribution $p(\mathbf{y}_n|\mathbf{x}_n, \mathcal{D}, m)$ for each datapoint n giving the **log-pointwise predictive density**:

$$LLPD(m) = \sum_{n=1}^{N} logp(\mathbf{y}_n|\mathbf{x}_n, \mathcal{D}, m) = \sum_{n=1}^{N} log \int p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}, m)p(\boldsymbol{\theta}|\mathcal{D}, m)d\boldsymbol{\theta} \approx \sum_{n=1}^{N} log(\frac{1}{S}\sum_{s=1}^{S} p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}_s, m)) \tag{108}$$

where $\boldsymbol{\theta}_s \sim p(\boldsymbol{\theta}|\mathcal{D}, m)$ is a posterior sample.

Problem: predicts nth data points using all the data, including $\mathbf{y}_n$.

We should compute the **expected log-pointwise density** on *future data* $(\mathbf{x}_*, \mathbf{y}_*)$.

Using leave out one approximation, and then monte carlo to approximate the integral:

$$ELPD(m) = \mathbb{E}_{x_*,y_*}[logp(\mathbf{y}_*|\mathbf{x}_*, \mathcal{D}, m)] \tag{109}$$

$$= \sum_{n=1}^{N} logp(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}, m)p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m)d\boldsymbol{\theta} \tag{110}$$

$$\approx \sum_{n=1}^{N} log(\frac{1}{S}\sum_{s=1}^{S} p(\mathbf{y}_n|\mathbf{x}_n|\boldsymbol{\theta}_{s,-n}, m)) \tag{111}$$

which requires computing N different posteriors $p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m)$, leaving one data point out at a time, which is slow.

A faster solution computes $p(\boldsymbol{\theta}|\mathcal{D}, m)$ once and then uses importance sampling to approximate the integral.

Indeed, if $f(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D}_{-n}, m)$ is the target distribution and $g(\boldsymbol{\theta}) = p(\boldsymbol{\theta}|\mathcal{D}, m)$ is the proposal, we can define the importance weight:

$$w_{s,-n} = \frac{f(\boldsymbol{\theta})}{g(\boldsymbol{\theta})} \propto \frac{1}{p(\mathcal{D}_n, \boldsymbol{\theta}_s)} \tag{112}$$

which we than normalize:

$$\hat{w}_{s,-n} = \frac{w_{s,-n}}{\sum_{s'=1}^{S} w_{s',-n}} \tag{113}$$

and estimate:

$$ELPD_{IS-LOO}(m) = \sum_{n=1}^{N} log(\sum_{s=1}^{S} \hat{w}_{s,-n} p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}_s, m)) \tag{114}$$

We can reduce the variance by fitting a pareto distribution to each set of weights for each sample, and use this to smooth the weights. This is called **Pareto smoothed importance sampling**.

Information Criteria

An alternative approach to cross validation is to score models using the negative log likelihood on the training set plus a *complexity penalty*:

$$\mathcal{L}(m) = -logp(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + C(m) \tag{115}$$

which is the **information criterion**.

Definition 54. Minimum description length

The goal is for the sender to communicate data to the receiver. First we need to encode the model m which takes $C(m) = -logp(m)$ bits. Then the receiver can fit the model, by computing $\hat{\boldsymbol{\theta}}_m$ and can thus approximately reconstruct the data. The sender also needs to send the residual errors that cannot be explained by the model, which takes an additional:

$$-L(m) = -logp(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) = -\sum_{n} logp(\mathbf{y}_n|\mathbf{x}_n, \hat{\boldsymbol{\theta}}, m) \tag{116}$$

bits.

Minimizing the sum of these terms is the **minimum description length principle**.

Definition 55. Bayesian information criterion has the form:

$$\mathcal{L}_{BIC}(m) = -2logp(\mathcal{D}|\hat{\boldsymbol{\theta}}, m) + D_m logN \tag{117}$$

where D_m is the **degrees of freedom** of model m which is derived from penalizing the hessian of the negative log joint $\mathbf{H} = \nabla\nabla logp(\mathcal{D}, \boldsymbol{\theta})$.

Probabilistic Graphical Models

Definition 56. Probabilistic Graphical Models provide a way of defining joint distribution on sets of random variables. Nodes represent random variables, and the lack of edges represent **conditional independence** assumptions between variables.

Directed Graphical Models

These are based on directed acyclic graphs, which admit a *topological ordering*, meaning nodes can be ordered such that parents come before children.

Definition 57. The **ordered markov property** is the assumption that *a node is conditional indepedent of all its predecessors*:

$$x_i \perp x_{pred(i)-pa(i)} | x_{pa(i)} \tag{118}$$

Hence the joint distribution factorizes:

$$p(x_1, ..., x_n) = p(x_1)p(x_2|x_1)p(x_3|x_2, x_1) \cdot ... \cdot p(x_n|x_1, .., x_{n-1}) = \prod_{i=1}^{N} p(x_i|x_{pa}(i)) \tag{119}$$

If we have n nodes taking K discrete states, the full joint requires $\mathcal{O}(K^n)$ parameters to specify, whereas the factorized joint requires $\mathcal{O}(nK^{n_p+1})$ where N_p is the maximum fan in of any node in the graph.

Conditional Independence Properties

Definition 58. We write $x_A \perp x_B | x_C$ if **A is conditionally independent of B given C in the graph we are considering**.

Let $I(G)$ be the set of all such conditional independence statements encoded by the graph, and $I(p)$ be the set of all such conditional independence statements that hold true in some distribution p. We say that G is an **independence map for p** or that **p is Markov with respect to p** $\iff I(G) \subset I(p)$.

WEe say that **G is a minimal independence map of p** if G is an independence map of p, and there is no $G' \subset G$ which is an independence map of p.

Definition 59. Global Markov Properties (d-seperation)

We say an **undirected path P is d-seperated** by a set of nodes C iff at least one of the following conditions hold:

1. P contains a chain or pipe $s \to m \to t$ where $m \in C$
2. P contains a test or fork $s \leftarrow m \to t$ where $m \in E$
3. P contains a v-structure $s \to m \leftarrow t$ where m is not in C and neither is any descendant of m

And a *set of nodes* A is **d-seperated** from a different set of nodes B given a third observed set C iff each undirected path from every node $a \in A$ to every node $b \in B$ is d-seperated by C

When the previous is true, it is an instance of the **global markov property**.

We can use the **Bayes ball algorithm** to detect d-seperation in a DAG.

Definition 60. The smallest set of nodes that renders a node i conditionally independent of all other nodes in a graph is called i's **Markov blanket**, denoted mb(i).

In a DPGM, the markov blanket is equal to the parents, the children, and the co-parents (other nodes who are also parents of its children).

Definition 61. The **local markov property** states that:

$$i \perp nd(i) - pa(i)|pa(i) \tag{120}$$

where nd(i) is the set of non-descendants of node i.

The **ordered markov property** states that:

$$i \perp pred(i) - pa(i)|pa(i) \tag{121}$$

Theorem 0.62. *Global markov property, local markov property and ordered markov property are equivalent in a DAG.*

Definition 63. Ancestor sampling generates prior samples from a DPGM by visiting the nodes in a topological ordering, parents before children, and sampling a value for each node given the parents.

Definition 64. Inference in graphical models is the process of computing the posterior over a set of **query nodes** Q given the observed values for a set of **visible nodes** V, while marginalizing over the irrelevant **nuissance variable** R (all nodes note queried or visible):

$$p_\theta(Q|V) = \frac{p_\theta(Q,V)}{p_\theta(V)} = \frac{\sum_R p_\theta(Q,V,R)}{p_\theta(V)} \tag{122}$$

The inference is in general NP-hard unless graph structures are of a special variety.

Learning the parameters

We are interested in the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ and assume the graph structure is fixed.

Typically we are fine with the posterior mode $\hat{\boldsymbol{\theta}} = argmaxp(\boldsymbol{\theta}|\mathcal{D})$ which might be reasonable since the parameters depend on all the data, rather than just a single data point.

Supose we have N local variables $\mathbf{x}_n, \mathbf{y}_n$ and 2 global variables (the parameters) shared across data samples.

We can write down the graphical model, and use the conditional independence properties to factorize the joint distribution, and hence the prior and likelihood, and hence the posterior. For example, the MLE can be computed via:

$$\hat{\boldsymbol{\theta}} = argmax_\theta \prod_{i=1}^{N} p(\mathcal{D}_i|\boldsymbol{\theta}_i) \tag{123}$$

We can place a dirichlet prior in our CPT's and perform map inference to help with zero-counts. In the presence of incomplete data, we typically resort to an optimization using expectation maximization, or stochastic gradient descent.

Markov Random Fields

When edge direction are unnatural to implement (ex. images), but we have local interactions, we can specify a **Markov Random field**.

- We gain symmetry, advantage of conditional random fields, but have less interpretable parameters, and require more computation to estimate these parameters.

We associate **potential functions** with each **maximal clique** in the graph, which is a set of nodes that are all neighbors of each other, and cannot be made any larger without losing the clique property.

A *potential* function associate to clique c is denoted $\psi_c(\mathbf{x}_c, \boldsymbol{\theta}_c)$ where $\boldsymbol{\theta}_c$ are the parameters, and ψ_c is any non-negative function.

Theorem 0.65 (Hammersley-Clifford)**.** *A joint distribution p that satisfies the conditional independence property implied by an undirected graph g can be factorized as:*

$$p(\boldsymbol{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{c \in C} \psi_c(\boldsymbol{x}_c, \boldsymbol{\theta}_c) \tag{124}$$

where ψ_c is the set of all maximal cliques of the graph, and $Z(\boldsymbol{\theta})$ is the ***partition function*** *that normalizes the density: $Z(\boldsymbol{\theta}) = \sum_x \prod_{c \in C} \psi_c(\boldsymbol{x}_c; \boldsymbol{\theta}_c)$*

We can rewrite the above factorization using an **energy based model**, which results in a **gibbs distribution** as:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} exp(-\mathcal{E}(\mathbf{x}; \boldsymbol{\theta})) \tag{125}$$

where $\mathcal{E}(\mathbf{x}) > 0$ is the **energuy of state x** given by $\mathcal{E}(\mathbf{x}; \boldsymbol{\theta}) = \sum_c \mathcal{E}(\mathbf{x}_c; \boldsymbol{\theta}_c)$.

Hopfield networks are fully connected Ising models with symmetric weight matrix $\mathbf{W} = \mathbf{W}^T$. The corresponding energy function has the form:

$$\mathcal{E}(\mathbf{x}) = -\frac{1}{2} \mathbf{x}^T \mathbf{W} \mathbf{x} \tag{126}$$

where $x_i \in \{-1, +1\}$.

The main application of Hopfield networks is *associative memory or content addressable memory.*

MRF's with latent variables: Boltzmann machines

These are a way to represent high dimensional joint distributions in discrete spaces.

A special case that supports efficient approximate inference is the **Restricted Boltzmann machine**, since the hidden ndoes re conditionally independent given the visible nodes $p(\mathbf{z}|\mathbf{x}) = \prod_{k=1}^{K} p(z_k|\mathbf{x})$. The architecture is arranged in two layers, and so that there are no connections between nodes within the same layer.

We can create **deep boltzmann machines** by stacking multiple layers.

Maximum Entropy Models

Recall that the exponential family is the distribution with maximal entropy subject to the constraint that the expected value of the features (sufficient statistics) $\phi(\mathbf{x})$ match the empirical expectations, leading to the model $p(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} exp(\boldsymbol{\theta}^T \phi(\mathbf{x}))$.

If the features $\phi(\mathbf{x})$ decompose according to a graph structure, we get a MRF called a **maximum entropy model**.

Gaussian MRF's

Definition 66. A **gaussian graphical model or Gaussian MRF** is a pariwise MRF of the form:

$$p(\mathbf{x}) = \frac{1}{Z(\boldsymbol{\theta})} \prod_{i \sim j} \psi_{ij}(x_I, x_j) \prod_i \psi_i(x_i) \tag{127}$$

$$\psi_{ij}(x_i, x_j) = exp(-\frac{1}{2} x_i \Lambda_{ij} x_j) \tag{128}$$

$$\psi_i(x_i) = exp(-\frac{1}{2} \Lambda_{ii} x_i^2 + \eta_i x_i) \tag{129}$$

$$Z(\boldsymbol{\theta}) = (2\pi)^{\frac{D}{2}} |\Lambda|^{-\frac{1}{2}} \tag{130}$$

The joint distribution takes the form:

$$p(\mathbf{x}) \propto exp[\boldsymbol{\eta}^T \mathbf{x} - \frac{1}{2} \mathbf{x}^T \boldsymbol{\Lambda} \mathbf{x}] \tag{131}$$

Conditional Independence Properties

Definition 67. Given 3 sets of nodes A, B, C, we say $\mathbf{X}_A \perp \mathbf{X}_B|\mathbf{X}_C \iff$ C separates A and B in the graph G. This is called the **global markov property**

In an MRF, a nodes markov blanket is its set of immediate neighbors. This is the **local markov property**

Two nodes are conditionally independent given the rest if there is no direct edge between them. This is called the **pairwise markov property**.

Theorem 0.68. *The three are equivalent as in DGM.*

Generation (sampling)

Unlike in DGM, there is no topological ordering, so sampling is slow. Furthermore, we cannot easily compute the probability of any configuration unless we know the normalizing constant Z. It is therefore common to us MCMC methods for generating from an MRF.

Learning

Computing MLE is computationally expensive even in fully observable case due to the partition function. Computing the posterior over parameters is even harder because of the evidence.

Conditional Random Fields

Definition 69. A **conditional random field** is a Markov random field defined on a set of related label nodes $\mathbf{y}$ whose joint probability is predicted conditional on a fixed set of input nodes $\mathbf{x}$. It corresponds to a model of the form:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z(\mathbf{x}, \boldsymbol{\theta})} \prod_c \psi_c(\mathbf{y}_c; \mathbf{x}, \boldsymbol{\theta}) \tag{132}$$

This is useful because they capture dependencies amongst the output labels, and can be used for **structured prediction** where the output $\mathbf{y}$ that we want to predict given the input $\mathbf{x}$ lives in some structured space.

Ex: in image segmentation, we can pass the output of a CNN which has locally smooth properties due to convolution into a fully connected CRF, which has connections between all pixels. This can capture long range connections which increase the sharpness of the segmentation boundaries.

Ex: in object detection, we can use sliding window dector with a model that classifies within an image patch. But this doesn't work when variance of occlusion and deform ability is high. Instead, we can subdivide into smaller patches and detect each independently. The CRF is used to enforce spatial coherence of the parts.

Factor Graphs

A **factor graph** is a graphical representation that unifies directed and undirected models.

Definition 70. A **standard factor graph** is an undirected bipartite graph with two kinds of nodes. Round nodes represent variables, square nodes represent factors, and there is an edge from each variable to every factor that mentions it.

Definition 71. A **Forney factor graph** is a graph in which nodes represent factors, and edges represent variables. These support hierarchical constructions, in which complex dependency structure between variables can be represented as a black box.

Structured Causal Models

A **structured casual model** is a triple $\mathcal{M} = (\mathcal{U}, \mathcal{V}, \mathcal{F})$ where $\mathcal{U} = \{U_i : i = 1, .., N\}$ is a set of unexplained or **exogenous** noise variables which are passed as input to the model, $\mathcal{V} = \{V_i : i = 1, .., N\}$ is a set of

endogenous variables that are part of the model itself, and $\mathcal{F} = \{F_i : i = 1, .., N\}$ is a set of deterministic function of the form $V_i = f_i(V_{pa_i, U_i}$.

We assume the equations can structured in a **recursive** way, and our model is **causally sufficient**, meaning that $\mathcal{U}, \mathcal{V}$ are all the causally relevant (maybe unobserved) factors. This is called the **causal Markov assumption**.

A **structural equation model** is a special case of a structural causal model in which all the functional relationships are linear, and the prior on the noise terms is Gaussian.

Counterfactual reasoning tries to answer the questions "what is the distribution over outcomes if I were to do something else, given that I already did something."

The difficulty lies in the **fundamental problem of causal inference**: we cannot simultaneously see both outcomes in a deterministic state.

Information Theory

KL Divergence

We are interested in quantifing how much information we have gained by updating our beliefs from a distribution p to a new distribution p' such that the metric is *continuous, non-negative, permutation invariant, monotonic for uniform distributions, satisfies the chain rule for probability.*

The KL-divergence **uniquely satisfies the desiderata.**

Definition 72. The **Kullback-Leibler divergence (KL divergence) or *relative entropy*** is defined as:

$$D_{KL}(p|q) = \int_x p(x) log(\frac{p(x)}{q(x)}) = \mathbb{E}_{x\sim p}[log(\frac{p}{q})] \tag{133}$$

Requirement: our original distribution of beliefs q has support everywhere the updated distribution does. (It would take infinite amount of information for us to update our beleifs in some outcome to change from being measure zero set, to non-zero)

Lemma 73. $D_{KL}(p||q) \geq 0$ *with equality iff* $p = q$.

Lemma 74. *KL divergence is invariant to reparameterizations.*

Lemma 75.

$$D_{KL}[p(x,y)|q(x,y)] = D_{KL}(p(x)|q(x)) + \mathbb{E}_{x\sim p}[D_{KL}(p(y|x)|q(y|x)] \tag{134}$$

Moment Projection

Suppose we compute q by minimizing the forward KL:

$$q = argmin_q D_{KL}(p|q) \tag{135}$$

this is called **M-projection** since the optimal q matches the moments of p, hence *moment matching.*

Information Projection

Suppose we compute q by minimizing the reverse KL:

$$q = argmin_q D_{KL}(q|p) \tag{136}$$

this is called **I-projection**

Theorem 0.76 (Compression Lemma)**.** *For any distributions P, Q with a well-defined KL divergence, and for any scalar function ϕ defined on the domain of the distributions:*

$$\mathbb{E}_P[\phi] \leq log\mathbb{E}_Q[e^\phi] + D_{KL}(P|Q) \tag{137}$$

Corollary 77. ***Donsker Varadhan Variational Representation of the KL**:*

$$D_{KL}(P|Q) = sup_\phi \mathbb{E}_P[\phi] - log\mathbb{E}_Q[e^\phi] \tag{138}$$

Data Processing Inequality for KL

We cannot increase the infromation gain from q to p by processing our data and measuring it.

Consider two distributions $p(x), q(x)$ and a probabilistic channel $t(y|x)$. If $p(y)$ is the distribution that results form sending samples from $p(x)$ through the channel $t(y|x)$ and similarly for $q(y)$ we have:

$$D_{KL}(p(x)|q(x)) \geq D_{KL}(p(y)|q(y)) \tag{139}$$

KL Divergence and MLE

Suppose we wish to approximate a true distribution p by M-projection:

$$q = argmin_q D_{KL}(p|q) \tag{140}$$

but suppose we only have samples from p. We can use the empirical distribution of p, which puts a probability atom on the observed training data, and zerom mass elsewhere:

$$p_{\mathcal{D}}(x) = \frac{1}{N_{\mathcal{D}}} \sum_{n=1}^{N_{\mathcal{D}}} \delta(x - x_n) \tag{141}$$

Then by the sifting property of delta functions:

$$D_{KL}(p_{\mathcal{D}}|q) = \frac{-1}{N_{\mathcal{D}}} \sum_n logq(x_n) + C \tag{142}$$

where $C = \int p_{\mathcal{D}}(x) log p_{\mathcal{D}}(x)$ is a constant independent of q.

We can rewrite:

$$D_{KL}(p_{\mathcal{D}}|q) = \mathbb{H}_{ce}(p_{\mathcal{D}}, q) - \mathbb{H}(p_{\mathcal{D}}) \tag{143}$$

where $H_{ce}(p, q) = -\sum_k p_k log q_k$ is the **cross entropy**, the average negative log likelihiood of q evaluated on the training set.

Minimizing the KL divergence to the empirical distribution is equivalent to minimizing the cross entropy to the empirical distirbution which is equivalent to minimizing likelihood

KL Divergence and Bayesian Inference

Consider a prior set of beliefs described by a joint $q(\theta, D) = q(\theta)q(D|\theta)$ invoplving the prior $q(\theta)$ and likelihood $q(D|\theta)$.

Suppose we observe a dataset D_0. We can search for the joint distribution that is as close as possible to our prior, but that respects the constraints of our observed data:

$$p(\theta, D) = argmin D_{KL}(p(\theta, D), q(\theta, D)) s.t. p(D) = \delta(D - D_0) \tag{144}$$

Writing the KL in chain form:

$$D_{KL}(p(\theta, D), q(\theta, D)) = D_{KL}(p(D), q(D)) + D_{KL}(p(\theta|D)|q(\theta|D)) \tag{145}$$

implies the solution is given by:

$$p(\theta, D) = p(D)p(\theta|D) = \delta(D - D_0)q(\theta|D) \tag{146}$$

and so our updated beliefs have a marginal over θ:

$$p(\theta) = \int p(\theta, D) dD = q(\theta|D = D_0) \tag{147}$$

KL divergence and exponential Families

The KL divergence between two exponential family distributions from the same family has a nice closed form.

Suppose $p(\mathbf{x}) = h(\mathbf{x}) exp[\boldsymbol{\eta}^T \mathcal{T}(\mathbf{x}) - A(\boldsymbol{\eta})]$ is an exponential family with natural parameter $\boldsymbol{\eta}$, base masure $h(\mathbf{x})$, sufficient statistics $\mathcal{T}(\mathbf{x})$ and the convex log partition function $A(\boldsymbol{\eta})$.

Lemma 78. *The KL divergence between two exponential family of the same form is:*

$$D_{KL}(p(\boldsymbol{x}|\boldsymbol{\eta}_1), p(\boldsymbol{x}|\boldsymbol{\eta}_2)) = (\boldsymbol{\eta}_1 - \boldsymbol{\eta}_2)^T \mathbb{E}_{\eta_1}[\mathcal{T}(\boldsymbol{x})] - A(\boldsymbol{\eta}_1) + A(\boldsymbol{\eta}_2) \tag{148}$$

For example, the KL divergence between two multivariate gaussians is:

$$D_{KL}(N(\mathbf{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)|N(\mathbf{x}|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) = \frac{1}{2}[tr(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1) + (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_2) - D + log(\frac{det\boldsymbol{\Sigma}_2}{det\boldsymbol{\Sigma}_1})] \tag{149}$$

Bregman Divergence

Let $f : \Omega \to \mathbb{R}$ be a C^1 strictly convex function defined on a closed convex set Ω. We define the **Bregman divergence** associated with f as:

$$B_f(\mathbf{w}|\mathbf{v}) = f(\mathbf{w}) - f(\mathbf{v}) - (\mathbf{w} - \mathbf{v})^T \nabla f(\mathbf{v}) = f(\mathbf{w}) - \hat{f}_v(\mathbf{w}) \tag{150}$$

where $\hat{f}_v(\mathbf{w}) = f(v) + (\mathbf{w} - \mathbf{v})^T \nabla f(\mathbf{v})$ is the first order taylor series approximation to f centered at $\mathbf{v}$.

- If $f(\mathbf{w}) = ||\mathbf{w}||^2$ then $B_f(\mathbf{w}||\mathbf{v}) = ||\mathbf{w} - \mathbf{v}||^2$ is the squared euclidean distance
- If $f(\mathbf{w}) = \mathbf{w}^T\mathbf{Q}w$ then $B_f(\mathbf{w}|\mathbf{v}) = (\mathbf{w} - \mathbf{v})Q^{-1}(\mathbf{w} - \mathbf{v})$ is the squared mahalnobis distance
- If $\mathbf{w}$ are the natural parameters for an exponential family and $f(\mathbf{w}) - logZ(\mathbf{w})$ is the log normalizer, then *the bregman divergence is the same as the KL*

Entropy

Definition 79. The **entropy** of a random variable with distribution p is defined as:

$$\mathbb{H}[p] = -\mathbb{E}_{x\sim p}[logp(x)] = -\int_x p(x)logp(x) \tag{151}$$

Lemma 80.

$$\mathbb{H}[p] = C + D_{KL}[p|unif] \tag{152}$$

Hence if p is uniform, the KL is zero, and the entropy achieves its maximal value. For discrete distribution with K states, the maximal entropy is $C = logK$

The differential entropy (continuous variable entropy) can be negative because a pdf can be bigger than 1

Note: differential entropy lacks the reparameterization independence of KL divergence. We pick of a factor given by the log determinant of the jacobian of the transformation.

Definition 81. The **typical set** of a probability distribution is the set whose elements have an information constant that is close to that of the expected information content from random samples of the distribution.

Suppose $p(\mathbf{x})$ is a distribution with support $\mathcal{X}$. The ε-typical set $\mathcal{A}_\varepsilon^N \in \mathcal{X}^N$ for $p(\mathbf{x})$ is the set of all length N sequences s.t.:

$$\mathbb{H}[p(\mathbf{x})] - \varepsilon \leq -\frac{1}{N}logp(\mathbf{x}_1, .., .\mathbf{x}_n) \leq \mathbb{H}[p(\mathbf{x})] + \varepsilon \tag{153}$$

If our samples are independent, the middle term is an N-sample empirical estimate of entropy. The **asymptotic equipartition property** states that this will converge in measure to true entropy as $N \to \infty$.

Mutual Information

KL divergence gives us a way to measure how similar two distributions are. Mutual information is a measure of how dependant two random variables are.

Definition 82. The **mutual information** between r.v. X, Y is:

$$I(X,Y) = D_{KL}(p(x,y), p(x)p(y)) = \int_{y\in Y}\int_{x\in X} p(x,y)log(\frac{p(x,y)}{p(x)p(y)}) \tag{154}$$

Independent random variables have no mutual information. Measures the information gain if we update from a model that treats the two variables as independet p(x)p(y) to one that models their true joint density p(x, y)

Lemma 83.

$$I(X,Y0 = \mathbb{H}(X) - \mathbb{H}(X|Y) = \mathbb{H}(Y) - \mathbb{H}(Y|X) \tag{155}$$

Measures the reduction in uncertainty about Y after observing X

Theorem 0.84 (Data Processing Inequality)**.** *Suppose we have an unknown variable X, and observe a noisy function of it, Y. If we process the noisy observations in some way to create a new variable Z, we cannot increase the amount of information we have about the unknown quantity X*

If $X \to Y \to Z$ is a Markov chain, so that $X \perp Z|Y$ then $I(X,Y) \geq I(X,Z)$

Corollary 85. *Suppose we have a chain $\theta \to X \to s(X)$, then $I(\theta, s(X)) \leq I(\theta, X)$.*

*If this holds with equality, then we say that $s(X)$ is a **sufficient statistic** of the data X for the process of inferring θ.*

*We say s is a **minimal sufficient statistic for X** if moreover $s(X) = f(s'(X))$ for some function f and all sufficient statistic $s'(X)$. In this case, it is sufficient for inferring θ, and contains no extra redundant information. Hence $s(X)$ maximally compresses the data X.*

Definition 86. To generalize MI to a set of random variables, we can define the **multi-information** or **total correlation**:

$$TC(\{X_1,...,X_d\}) = D_{KL}(p(\mathbf{x}|\prod_d p(x_d)) = \int_{\mathbf{x}} p(\mathbf{x})log(\frac{p(\mathbf{x})}{\prod_d p(x_d)}) = \sum_d \mathbb{H}(x_d) - \mathbb{H}(\mathbf{x}) \tag{156}$$

Definition 87. The conditional mutual information can be used to give an inductive definition of the **multiple mutual information** or **co-information**:

$$I(X_1,...,X_D) = I(X_1,...,X_{D-1}) - I(X_1,...,X_{D-1}|X_D) \tag{157}$$

Lemma 88.

$$I(X_1,..,X_D) = -\sum_{\mathcal{T}\subset\{1,...,D\}} (-1)^{|\mathcal{T}|}\mathbb{H}(\mathcal{T}) \tag{158}$$

Variational bounds on mutual information

Lemma 89. Upper bound

Suppose that the joint $p(\boldsymbol{x}, \boldsymbol{y})$ is intractable to evaluate, but we can sample from $p(\boldsymbol{x})$ and evaluate the conditional $p(\boldsymbol{y}|\boldsymbol{x})$. Also assume we can approximate $p(\boldsymbol{y})$ by $q(\boldsymbol{y})$

Then:

$$I(\boldsymbol{x}, \boldsymbol{y}) \leq E_{x\sim p}[D_{KL}(p(\boldsymbol{y}|\boldsymbol{x})|q(\boldsymbol{y})] \tag{159}$$

The bound is tight if $q(\mathbf{y}) = p(\mathbf{y})$

Lemma 90. BA Lower Bound

Suppose that the joint $p(\boldsymbol{x}, \boldsymbol{y})$ is intractable to evaluate, but we cna evaluate $p(\boldsymbol{x})$. Also assume we can approximate $p(\boldsymbol{x}|\boldsymbol{y})$ by $q(\boldsymbol{x}|\boldsymbol{y})$.

Then:

$$I(\boldsymbol{x}, \boldsymbol{y}) \geq E_{(x,y)\sim p(x,y)}[logq(\boldsymbol{x}|\boldsymbol{y})] + h(\boldsymbol{x}) \tag{160}$$

InfoNCE Lower Bound

$$I_{NCE} = \mathbb{E}[\frac{1}{K}\sum_{i} = 1^{K} log \frac{e^{f(\boldsymbol{x}_i, \boldsymbol{y}_i)}}{\frac{1}{K}\sum_{j=1}^{K} e^{f(\boldsymbol{x}_j, \boldsymbol{y}_j)}}] \tag{161}$$

where $q(\boldsymbol{x}|\boldsymbol{y}) = \frac{p(\boldsymbol{x})e^{f(\boldsymbol{x},\boldsymbol{y}}}{Z(\boldsymbol{y})}$ and we do not require a tractable normalized distribution.

Data Compression (Source coding)

This is at the heart of information theory and probabilistic machine learning. *If we can model the probability of different kinds of data samples, then we can assign short **code words*** to the most frequently occurring ones, reserving longer encoding for the less frequent ones.

Lossless Compression **Lossless compression** is compressing the data in such a way that we can uniquely recover the original data. *Note: discrete data can always be losslessly compressed.*

Theorem 0.91 (Source Coding Theorem)**.** *The expected number of bits needed to lossless encode some data from distribution p is $\mathbb{H}(p)$.*

If we use any other model q, it will take excess bits since $H_{ce}(p, q) \geq \mathbb{H}(p)$. Equality is achieved if we add $D_{KL}(p, q)$ to the right side, and hence this is the extra number of bits we used.

- *Huffman coding and arithmetic coding are examples of lossless compression algorithms. The input is a probability distribution over strings.*

Lossy Compression To encode real valued signals, such as images and sound, we first have to quantize the signal into a sequence of symbols. A simple way to do this is **vector quantization** which allows the modeling of probability density function by the distribution of prototype vectors, by dividing a large set of points (vectors) into groups having approximately the same number of points closest to them. Each group is represented by its centroid, as in k-means.

Theorem 0.92 (The Rate Distortion Tradeoff)**.** *There is a tradeoff between the size of the representation (the number of symbols we use), and the resulting error.*

Assume we have a stochastic encoder $p(\boldsymbol{z}|\boldsymbol{x})$ and a stochastic decoder $d(\boldsymbol{x}|\boldsymbol{z})$, and a prior marginal $m(\boldsymbol{z})$.

*The **distortion** of the encoder-decoder pair is:*

$$D = -\int p(\boldsymbol{x})d\boldsymbol{x} \int logd(\boldsymbol{d}|\boldsymbol{z})e(\boldsymbol{z}|\boldsymbol{x})d\boldsymbol{z} \tag{162}$$

Note: if the decoder is a deterministic model plus Gaussian noise, and the encoder is deterministic, then:

$$d(\boldsymbol{x}|\boldsymbol{z}) = N(\boldsymbol{x}|f_d(\boldsymbol{z}, \sigma^2) \tag{163}$$

$$e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - f_e(\boldsymbol{x}) \tag{164}$$

and the distortion reduces to the ***reconstruction error***

$$D = \frac{1}{\sigma^2}\mathbb{E}_{x\sim p}[||f_d(f_e(\boldsymbol{x})) - \boldsymbol{x}||^2] \tag{165}$$

The ***rate*** *of our model is defined as:*

$$R = \mathbb{E}_{x\sim p}[D_{KL}(e(\boldsymbol{z}|\boldsymbol{x}))|m(\boldsymbol{z})] \tag{166}$$

which is the average kl between our encoding distribution and the marginal.

If we encode using $\boldsymbol{x}$ as our best representation, we would incur no distorion (and hence maximize likelihood), but it would incur a high rate, since there would be no compression. Conversely, if we take $e(\boldsymbol{z}|\boldsymbol{x}) = \delta(\boldsymbol{z} - 0)$ the encoder would ignore the input, and hence the rate would be 0, but the distortion would be high.

Bits Back Coding

We penalize in terms of the *excess bits* needed to encode our data, which is the cross entropy minus the entropy (the expected numbers of bits we need to encode minus the minimum number of of bits needed to encode).

How come we don't have to pay for the actual (total number of bits we use, i.e. the cross entropy?

Bits back coding is the process of in principle getting the bits needed by the optimal code given back to us.

- Alice sends data to bob. They a priori share encoder, marginal and decoder
- Alice sends a **two part code**
  1. Alice sends a sample code $z \sim p(z|x)$ from her encoder through a channel designed to efficiently encode samples from the marginal $m(z)$. This costs $-log_2 m(z)$ bits
  2. Alice uses here decoder $d(z|x)$ to comptute the residual error, and losslessly send that to Bob. This costs $-log_2 d(x|z)$ bits.
- the expected total number of bits required is $E_{x\sim p}[-log_2 d(x|z) - log_2 m(z)] = D + \mathbb{H}_{CE}(p(z|x), m(z))$

We can get the bits back to convert the cross entropy term to a rate term.

- Bob can use the code z and the residual error to perfectly reconstruct x
- Bob also knows what specific code Alice sent, z, as well as what encoder she used $p(z|x)$
- When Alice drew the sample code $z \sim p(z|x)$ she needed some entropy source to generate the random sample.
- Bob can reverse engineer all of these sampling bits and thus recover the entropy source
- **Thus Alice can use the extra randomness in the choice of z to share more information**

Error Correcting Codes (Channel Coding)

Goal: *add redundancy to a signal x (which is the result of encoding the original data), such that when it is sent over to the receiver via a noisy transmission line, the receiver can recover from any corruptions that might occur to the signal*

Let $x \in \{0,1\}^m$ be the source message, where m is called the **block length**. Let **y** be the result of sending **x** over a **noisy channel**, a corrupted version of the message.

- Each message bit may be flipped independently with probability α

- Then $p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{m} p(y_i|x_i)$, $p(y_i|x_i = 0) = [1-\alpha, \alpha$ and $p(y+i|x_i = 1) = [\alpha, 1-\alpha]$

- We may assume Gaussian noise: $p(y_i|x_i = b) = N(y_i|\mu_b, \sigma^2)$

The receivers goal is to infer the true message from the noisy observations: to compute $argmax_x p(\mathbf{x}|\mathbf{y})$.

A common way to increase the chance of being able to recover the original signal is to add **parity check bits** before sending. These are deterministic functions of the original signal which specify if the sum of the inputs is odd or even.

The Information Bottleneck

Work with discriminiative models $p(\mathbf{y}|\mathbf{x})$ that us a *stochastic bottleneck* between the input $\mathbf{x}$ and the output $\mathbf{y}$ to prevent over fitting, improve robustness and calibration

Vanilla IB

Definition 93. We say **z is a representation of x** if $\mathbf{z}$ is a (possible stochastic) function of $\mathbf{x}$, and hence can be described by the conditional $p(\mathbf{z}|\mathbf{x})$.

We say that **representation z of x is sufficient for task y** if $\mathbf{y} \perp \mathbf{x}|\mathbf{z}$ or equivalently $\mathbb{I}(z; y) = \mathbb{I}(x, y)$.

We say **representation z of x is minimal sufficient statistic** if $\mathbf{z}$ is sufficient and there is no other $\mathbf{z}$ with smaller $\mathbb{I}(\mathbf{z}; \mathbf{x})$ value.

Objective is therefore:

$$min \beta \mathbb{I}(\mathbf{z}; \mathbf{x}) - \mathbb{I}(\mathbf{z}; \mathbf{y}) \tag{167}$$

where $\beta \geq 0$ is a lagrange multiplier, and we optimize w.r.t. to the distribution $p(\mathbf{z}|x)$ and $p(\mathbf{y}|\mathbf{z})$.

This is called the **information bottleneck principle**.

Variational IB

The **variational information bottleneck** is a variational upper bound on the optimization problem above.

Let $e(\mathbf{z}|\mathbf{x}) = p(\mathbf{z}|\mathbf{x})$ be the encoder, $b(\mathbf{z}|\mathbf{y}) \approx p(\mathbf{z}|\mathbf{y})$ be the backwards encoder, $d(\mathbf{z}|\mathbf{y}) \approx p(\mathbf{z}|\mathbf{y})$ be the decoder, and $m(\mathbf{z}) \approx p(\mathbf{z})$ be the marginal.

We have that:

$$\mathbb{I}(\mathbf{z}; \mathbf{y}) \geq \mathbb{E}_{(y,z)\sim p(y,z)}[log d(\mathbf{y}|\mathbf{x})] \tag{168}$$

we can approximate this expectation by sampling from:

$$p(\mathbf{y}, \mathbf{z}) = \int p(\mathbf{x}) p(\mathbf{y}|\mathbf{x}) p(\mathbf{z}|\mathbf{x}) d\mathbf{x} = \int p(\mathbf{x}, \mathbf{y}) e(\mathbf{z}|\mathbf{x}) d\mathbf{x} \tag{169}$$

Likewise:

$$\mathbb{I}(\mathbf{z}; \mathbf{x}) \leq \mathbb{E}_{(x,z)\sim p(x,z)}[log(\mathbf{z}|\mathbf{x})] - \mathbb{E}_{z\sim p(z)}[m(\mathbf{z})] \tag{170}$$

which we can approximate by sampling from $p(\mathbf{x}, z) = p(\mathbf{x}) p(\mathbf{z}|\mathbf{x})$.

Together, we can the upper bound on the IB objective:

$$\begin{aligned} \beta \mathbb{I}(\mathbf{z}; \mathbf{x}) - \mathbb{I}(\mathbf{z}; \mathbf{y}) &\leq \beta(\mathbb{E}_{(x,z)\sim p(x,z)}[log(\mathbf{z}|\mathbf{x})] - \mathbb{E}_{z\sim p(z)}[m(\mathbf{z})]) - \mathbb{E}_{(y,z)\sim p(y,z)}[log d(\mathbf{y}|\mathbf{x})] =: \mathcal{L}_{VIB} & (171) \\ &= -\mathbb{E}_{(x,y)\sim p_D(x) e(z|x) d(y|z)}[log d(\mathbf{y}|\mathbf{x})] + \beta \mathbb{E}_{p_D(x)}[D_{KL}(e(\mathbf{z}|\mathbf{x})|m(\mathbf{z})] & (172) \end{aligned}$$

we can take stochastic gradients f this objective and minimize it wrt to the paramterization of the encoder, decoder and marginal.

Optimization

Goal:

$$\boldsymbol{\theta}^* \in argmin_{\theta \in \Theta} \mathcal{L}(\boldsymbol{\theta}) \tag{173}$$

where $\mathcal{L} : \Theta \to \mathbb{R}$ is the scalar objective or loss function, and Θ is the parameter space we are optimizing over.

Automatic Differentiation

Autodiff is the means of automatically computing the partial derivatives of functions expressed as a composition of an arbitrary number of basic operations, such as in deep neural networks.

Let $\{e_1, ..., e_n\}$ be the standard basis for $\mathbb{R}^n$.

Recall

- linear map $F : \mathbb{R}^n \to \mathbb{R}^m$ can be represented as a matrix in $\mathbb{R}^{m \times n}$ whose columns are $F[e_1], .., F[e_n]$.
- We can write $T : \mathbb{R}^n \times ...\mathbb{R}^n \to \mathbb{R}^m$ to be a k-linear map, which can be represented as a tensor in $\mathbb{R}^{m \times n \times ... \times n}$. We denote by $T[\mathbf{x}_1, ..., \mathbf{x}_k] \in \mathbb{R}^m$ the application of such a k-linear map to a set of k vectors $\mathbf{x}_i \in \mathbb{R}^n$.
- The **derivative operator** is defined for an open set $U \subset \mathbb{R}^n$ and a differentiable function $f : U \to \mathbb{R}^m$:

  $$\partial f : U \to \mathcal{L}(\mathbb{R}^n, \mathbb{R}^m) \tag{174}$$

  which maps a point $\mathbf{x} \in U$ ot the Jacobian of all partial derivatives evaluted at $\mathbf{x}$
- When $m = 1$ the map ∂ recover the standard gradient ∇
- In the expression $\partial f(\mathbf{x})[\mathbf{v}]$ we refer to the argument $\mathbf{x}$ as the **linearization point** for the jacobian, and $\mathbf{v}$ as the perturbation
- We call the map $(\mathbf{x}, \mathbf{v}) \mapsto \partial f(\mathbf{x})[\mathbf{v}]$ the **Jacobian-vector product (JVP)**
- We call the map $(\mathbf{x}, \mathbf{u}) \mapsto \partial f(\mathbf{x})^T[\mathbf{u}]$ the **vector-Jacobian product (VJP)**
- If the function $f \in C^\infty(U)$ we can take further derivatives for example

  $$\partial^2 f : U \to \mathcal{L}(\mathcal{L}(\mathbb{R}^n, \mathbb{R}^n), \mathbb{R}^m) \tag{175}$$

  is a bilinear map representing all second-order partial derivatives. We can write $\partial^2 f(\mathbf{x})[e_i, e_j]$
- $\partial^k = \partial \circ \partial \circ ...\partial$
- when $k = 1, m = 1$ then the map $\partial^k f(\mathbf{x})$ corresponds to the Hessian matrix at any $\mathbf{x} \in U$
- Taylor series approximation:

  $$f(\mathbf{x} + \mathbf{v}) \approx f(\mathbf{x}) + \partial f(\mathbf{x})[\mathbf{v}] + \frac{1}{2!}\partial^2 f(\mathbf{x})[\mathbf{v}, \mathbf{v}] + ... + \frac{1}{k!}\partial^k f(\mathbf{x})[\mathbf{v}, ..., \mathbf{v}] \tag{176}$$

- A multi-input function $g : U \times V \to \mathbb{R}^m$ where $U \subset \mathbb{R}^{n_1}, V \subset \mathbb{R}^{n_2}$ is isomorphic to a uni-variable input on an open set $W \subset \mathbb{R}^{n_1 + n_2}$ and hence we can break the derivative maps into block form.
- If $f = g \circ h$ for $h : \mathbb{R}^n \to \mathbb{R}^p$ and $g : \mathbb{R}^p \to \mathbb{R}^m$ then the **chain rule** oberves that:

  $$\partial f(\mathbf{x}) = \partial g(h(\mathbf{x})) \circ \partial h(\mathbf{x}) \tag{177}$$

- **Fan out** involves several sub-expressions functions of the same input.

- f(**x**) = g(a(**x**), b(**x**)) $\implies \partial f(\mathbf{x}) = \partial_1 g(a(\mathbf{x}), b(\mathbf{x})) \circ \partial a(\mathbf{x}) + \partial_2 g(a(\mathbf{x}), b(\mathbf{x})) \circ \partial b(\mathbf{x})$

- **Composition** involves several sub-expression functions of different input
  - f(**x**, **y**) = g(a(**x**), b(**y**)) $\implies \partial_1 f(\mathbf{x}, \mathbf{y}) = \partial_1 g(a(\mathbf{x}), b(\mathbf{y})) \circ \partial a(\mathbf{x})$ and $\partial_2 f(\mathbf{x}, \mathbf{y}) = \partial_2 g(a(\mathbf{x}), b(\mathbf{y})) \circ \partial b(\mathbf{y})$

Differentiating Chains

Given a function $f : U \subset \mathbb{R}^n \to \mathbb{R}^m$ and a linearization point $\mathbf{x} \in U$, auto diff computes either:

- the JVP $\partial f(\mathbf{x})[v]$ for an input perturbation $\mathbf{v} \in \mathbb{R}^n$
- the VJP $\partial f(\mathbf{x})^T[u]$ for an output perturbation $\mathbf{u} \in \mathbb{R}^m$

Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$ is a composition $f = c \circ b \circ a$.

Then by the chain rule, we can consider the JVP against an input perturbation $\mathbf{v} \in \mathbb{R}^n$:

$$\partial f(\mathbf{x})[\mathbf{v}] = \partial c(b(a(\mathbf{x})[\partial b(a(\mathbf{x}))[\partial a(\mathbf{x})[\mathbf{v}]] \tag{178}$$

This *right to left evaluation order corresponds to* ***forward-mode automatic differentiation***.

Given as input $f : \mathbb{R}^n \to \mathbb{R}^m$ as a chain $f = f_T \circ ... \circ f_1$, a linearization point $\mathbf{x} \in \mathbb{R}^n$ and input perturbation $\mathbf{v} \in \mathbb{R}^n$

```
def forward_mode_auto_diff_chain(f, x, v):
    x_0, v_0 = x, v
    for t = 1, ..., T:
        x_t = f_t(x_tm1)
        v_t = partial(f_t(x_tm1))[v_tm1]
```

outputs $x_T = f(\mathbf{x}), v_T = \partial f(\mathbf{x})[\mathbf{v}]$

By contract, we can transpose and consider the VJP against an output perturbation $\mathbf{u} \in \mathbb{R}^m$:

$$\partial f(\mathbf{x})^T[\mathbf{u}] = \partial a(\mathbf{x})^T[\partial b(a(\mathbf{x}))^T[\partial c(b(a(\mathbf{x})))^T[u]]] \tag{179}$$

and now the backwards ordered Jacobian evaluation corresponds to **reverse-mode automatic differentiation.**

```
def backward_mode_auto_diff_chain(f, x, u):
    x_0 = x
    for t = 1, .., T:
        x_t = f_t(x_tm1)
    u_T = u
    for t = T, ..., 1:
        u_tm1 = partial(f_t(x_tm1))^T[u_t]
```

outputs $x_T = f(\mathbf{x}), u_0 = \partial f(\mathbf{x})^T[\mathbf{u}]$

Note: reverse mode autodiff is faster than forward-mode when the output is scalar valued, such as a scalar loss.

Note: reverse mode autodiff stores all the chain prefixes before its backwards traversal, so it consumes more memory than forward mode.

Differentiating Circuits

When primitives can accept multiple inputs, we can extend to directed acylic graphs over primitive operations, or **computation graphs**.

The **input nodes** of a graph symbolize function arguments, and **primitive nodes** are those labelled by a primitive operation.

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ composing $f_1, ..., f_T$ in a **topological order** where f_1 is the identity, a linearization point $\mathbf{x} \in \mathbb{R}^n$ and input perturbation $\mathbf{v} \in \mathbb{R}^n$.

```
def forward_mode_auto_diff(f, x, v):
    x_1, v_1 = x, v
    for t = 2, ..., T:
        [q1, ..., qr] = Pa(t)
        x_t = f_t(x_q1, ..., x_qr)
        v_t = sum[partial_i(f_t(x_q1, ..., x_qr))[v_qi] for i in 1, ..., r]
```

outputs $x_T = f(\mathbf{x}), v_T = \partial f(\mathbf{x})[\mathbf{v}]$

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ composing $f_1, ..., f_T$ in a **topological order** where f_1, f_T is the identity, a linearization point $\mathbf{x} \in \mathbb{R}^n$ and perturbation $\mathbf{u} \in \mathbb{R}^m$.

```
def reverse_mode_auto_diff(f, x, u):
    x_1 = x
    for t = 2, ..., T:
        [q1, ..., qr] = Pa(t)
        x_t = f_t(x_q1, ..., x_qr)
    u_Tm1_T = u
    for t = T-1, ..., 2:
        [q1, ..., qr] = Pa(t)
        u_tt = sum[u_t_c for c in Ch(t)]
        u_qi_t = partial_i(f_t(x_q1, ..., x_qr))^Tu_tt for i = 1, ..., r
```

outputs $x_T = f(\mathbf{x}), u_{1_2} = \partial f(\mathbf{x})^T[\mathbf{u}]$

Stochastic Gradient Descent

We consider optimizers for unconstrained differentiable objectives. Gradient based solvers which perform iterative updates of the form:

$$\boldsymbol{\theta} t + 1 = \boldsymbol{\theta}_t - \eta_t \mathbf{C}_t \mathbf{g}_t \tag{180}$$

where $\mathbf{g}_t = \nabla \mathcal{L}(\boldsymbol{\theta}_t)$ is the gradient of the loss, and $\mathbf{C}_t$ is an optional **conditioning matrix**.

If we set $\mathbf{C}_t = I$, the method is known as **steepest descent or gradient descent**.

If we set $\mathbf{C}_t = \mathbf{H}_t^{-1}$ where $\mathbf{H}_t = \nabla^2 \mathcal{L}(\boldsymbol{\theta}_t)$ is the Hessian, we get **Newtons method.**

In most cases, we cannot compute the exact gradient, either because the loss is stochastic, or because we approximate the loss by randomly subsampling data. In this case, we can modify the above to get an unbiased approximation. We can approximate the gradient using a minibatch $\mathcal{B}_t$ of size $B = |\mathcal{B}_t|$:

$$\hat{\mathbf{g}}_t = \hat{\nabla} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{n \in \mathcal{B}_t} \nabla \mathcal{L}_n(\boldsymbol{\theta}_t) \tag{181}$$

Inserting this into the iterative update yields **stochastic gradient descent.**

Natural Gradient Descent

Natural gradient descent is a second order method for optimizing the parameters of a conditional probability distribution $p_\theta(\mathbf{y}|\mathbf{x})$. The key idea is the compute parameter updates by measuring the distances between the induced distributions rather than comparing parameter values directly. NGD can converge much faster than other gradient methods.

Definition

We can approximate the KL divergence in terms of the fisher informaition matrix, so for any given input $\mathbf{x}$:

$$D_{KL}(p_\theta(\mathbf{y}|\mathbf{x})|p_{\theta+\delta}(\mathbf{y}|\mathbf{x})) \approx \frac{1}{2}\boldsymbol{\delta}^T\mathbf{F}_x\boldsymbol{\delta} \tag{182}$$

where the FIM is:

$$\mathbf{F}_x(\theta) = -\mathbb{E}_{p_\theta(y|x)}[\nabla^2 log p_\theta(\mathbf{y}|\mathbf{x})] = \mathbb{E}_{p_\theta(\mathbf{y}|\mathbf{x})}[(\nabla log p_\theta(\mathbf{y}|\mathbf{x})(\nabla log p_\theta(\mathbf{y}|\mathbf{x}))^T] \tag{183}$$

We can compute the average KL between the current and updated distributions by taking $F(\boldsymbol{\theta}) := \mathbb{E}_{p_{\mathcal{D}(x)}[\mathbf{F}_x(\boldsymbol{\theta})]}$

NGD uses the FIM as a preconditioning matrix, yielding:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t\mathbf{F}(\boldsymbol{\theta}_t)^{-1}\mathbf{g}_t \tag{184}$$

and the term $\mathbf{F}(\boldsymbol{\theta}_t)^{-1}\mathbf{g}_t = F^{-1}\nabla\mathcal{L}(\boldsymbol{\theta}_t)$ is the **natural gradient**.

NGD as trust region

We can interpret standard gradient descent as optimizing a linear approximation to the objective subject to a penalty on the l_2 norm of the change in parameters, so if $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\delta}$, we optimize:

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \mathbf{g}_t^T\boldsymbol{\delta} + \eta||\boldsymbol{\delta}||_2^2 \tag{185}$$

If we replace the squared distance with the squared FIM distance $||\boldsymbol{\delta}||_F^2 = \boldsymbol{\delta}^T\mathbf{F}\boldsymbol{\delta}$, this is equivalent to the squared norm in a new coordinate system, and we recover the natural gradient update.

Note: If our conditional distribution is an exponential family, NGD is identical to generalized gauss newton method, and in the online setting, equivalent to performing sequential Bayesian inference using the extended Kalman filter.

Benefits

- The FIM is always positive definite. The hessian can have negative eigenvalues at saddle poionts, which are prevalent in high dimensions
- Its easy to approximate F online from minibatches, since its an expectation of outer products of gradient vectors, whereas the hessian is sensitive to noise introduced by minibatch approximation
- The trust region optimization performed by NGD updates parameters in a way that matter most for prediction, which allows taking larger steps in uninformative regions of parameter space, thereby avoiding getting stuck in plateaus
- FIM is invariant to paramaterization, so helps with complex models, and overdetermined systems

Note: for exponential families, SGD on moment parameters is equivalent to NGD on natural parameters

Approximating the natural gradient

The cost we pay is computing the inverse of the fisher information matrix. Some speed ups include:

- Assuming its diagonal

- Low rank block diagonal approximation
- Cholesky factorized block matrix
- **KFAC**: kronecker-factorized approximate curvature, which is a block diagonal matrix where each block is a kronecker product of two small matrices
- Approximate the Fim by replacing the model's distribution with the empirical one, yielding the **empirical fisher**

$$F = \mathbb{E}_{p_\theta(\mathbf{x},\mathbf{y})}[\nabla logp(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})\nabla logp(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})^T] \tag{186}$$

$$\approx \mathbb{E}_{p_D(x,y)}[\nabla logp(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})\nabla logp(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})^T] \tag{187}$$

$$= \frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}} \nabla logp(\mathbf{y}|\mathbf{x}|\boldsymbol{\theta})\nabla logp(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})^T \tag{188}$$

Note: when we reach a flat part of parameter space, the empirical fisher becomes singular, and hence the algorithm gets stuck. The true fisher takes expectations over the outputs, marginalizing out $\boldsymbol{y}$*, allowing us to detect small changes in the output.*

- Use exact computation of $\mathbf{F}$, but solve for $\mathbf{F}^{-1}\mathbf{g}$ approximately using **conjugate gradient methods**. This is is called **Hessian free optimization**

Gradients of Stochastic Functions

Goal is to compute the gradient of stochastic functions of the form:

$$\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_\psi(z)}[l(\boldsymbol{\psi},\mathbf{z}] \tag{189}$$

MiniBatch Approximation

In the simplest case, $q_\psi(z)$ does not depend on ψ. Then we can push gradients inside the expectation:$\nabla\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}[\nabla l(\boldsymbol{\psi},\mathbf{z})]$ and perform a monte carlo sample procedure on $\mathbf{z}$ to approximate the gradient.

Optimizing parameters of a distribution

If the stochasticity depends on the parameters we are optimizing, for example $\mathbf{z}$ can be an action sampled from a stochastic policy q_ψ as in RL, then the gradient is given by:

$$\nabla_\psi \mathbb{E}_{q_\psi(z)}[l(\boldsymbol{\psi},\mathbf{z})] = \nabla_\psi \int l(\boldsymbol{\psi},\mathbf{z})q_\psi(\mathbf{z})d\mathbf{x} \tag{190}$$

$$= \int [\nabla_{\boldsymbol{\psi}} l(\boldsymbol{\psi},\mathbf{z})]q_\psi(\mathbf{z})d\mathbf{z} + \int l(\boldsymbol{\psi},\mathbf{z})[\nabla_\psi q_\psi(\mathbf{z})]d\mathbf{z} \tag{191}$$

We can monte carlo sample to approximate the first term:

$$\int [\nabla_{\boldsymbol{\psi}} l(\boldsymbol{\psi},\mathbf{z})]q_\psi(\mathbf{z})d\mathbf{z} \approx \frac{1}{S}\sum_{s=1}^{S} \nabla_\psi l(\boldsymbol{\psi},\mathbf{z}_s) \tag{192}$$

where $\mathbf{z}_s \sim q_\psi$.

We cannot use vanilla monte carlo sampling to approximate the integral, since we need to take gradients of the distribution itself. There are two main methods discussed below.

1. Score function estimator
2. Reparameterization trick

Score function estimator (likelihood ratio trick)

$$\nabla_\psi q_\psi(\mathbf{z}) = q_\psi(\mathbf{z}) \nabla_\psi log q_\psi(\mathbf{z}) \tag{193}$$

And hence the second term becomes:

$$\int l(\boldsymbol{\psi}, \mathbf{z}) [\nabla_\psi q_\psi(\mathbf{z})] d\mathbf{z} = \int l(\boldsymbol{\psi}, \mathbf{z}) q_\psi(\mathbf{z}) \nabla_\psi log q_\psi(\mathbf{z}) d\mathbf{z} = \mathbb{E}_{q_\psi(z)}[l(\boldsymbol{\psi}, \mathbf{z}) \nabla_\psi log q_\psi(\mathbf{z})] \tag{194}$$

which is the **score function estimator, since we are taking the gradient of a log probability distribution**. It is also called the **likelihood ratio gradient estimator** or **reinforce estimor**.

Now we can approximate with monte carlo:

$$\mathbb{E}_{q_\psi(z)}[l(\boldsymbol{\psi}, \mathbf{z}) \nabla_\psi log q_\psi(\mathbf{z})] \approx \frac{1}{S} \sum_{s=1}^{S} l(\boldsymbol{\psi}, \mathbf{z}_s) \nabla_\psi log q_\psi(\mathbf{z}_s) \tag{195}$$

The score function estimator usually has high variance. One way to reduce this is to use **control variates** where we replace $l(\boldsymbol{\psi}, \mathbf{z})$ by:

$$\hat{l}(\boldsymbol{\psi}, \mathbf{z}) = l(\boldsymbol{\psi}, \mathbf{z}) - c(b(\boldsymbol{\psi}, \mathbf{z}) - \mathbb{E}[b(\boldsymbol{\psi}, \mathbf{z})]) \tag{196}$$

where $b(\boldsymbol{\psi}, \mathbf{z})$ is a **baseline function** that is correlated with $l(\boldsymbol{\psi}, \mathbf{z})$ and $c > 0$ is a coefficient. Note that this estimator has the same expectation (unbiased gradients), but has lower variance.

Reparameterization trick

The score function estimator can have high variance, even when using a control variate. The reparameterization trick is a lower variance estimator, which can be applied if $l(\boldsymbol{\psi}, \mathbf{z})$ is differentiable w.r.t $\mathbf{z}$. We additionally require we can compute a sample from $q_\psi(\mathbf{z})$ by first sampling ε from some noise distribution q_0 that is independent of $\boldsymbol{\psi}$, and then transforming to $\mathbf{z}$ using a deterministic and differentiable function $z = r(\boldsymbol{\psi}, \varepsilon)$.

- $\mathbf{z} \sim \mathcal{N}(\mu, \sigma^2)$ is equivalent to the procedure
- $\varepsilon \sim \mathcal{N}(0, 1)$, and then $z = \mu + \sigma\varepsilon$

In this case, our stochastic objectives is:

$$\mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_\psi(z)}[l(\boldsymbol{\psi}, \mathbf{z})] = \mathbb{E}_{q_0(\varepsilon)}[l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon))] \tag{197}$$

and since $q_0(\varepsilon)$ is independent of $\boldsymbol{\psi}$, we can psuh the gradient inside the expectation, and approximate with monte carlo:

$$\nabla_\psi \mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{\nabla_\psi q_0(\varepsilon)}[l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon))] \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_\psi l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon_s)) \tag{198}$$

where $\varepsilon_s \sim q_o$. This is called the **reparameterization gradient or pathwise derivative**.

To compute $\nabla_\psi l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon_s))$ we need the **total derivative** since the function l depends on $\boldsymbol{\psi}$ directly, **and** via a noise sample.

Thus:

$$\nabla_\psi l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon_s)) = \nabla_\psi l(\boldsymbol{\psi}, \mathbf{z}) + \mathbf{J}^T \nabla_z l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon)) \tag{199}$$

where $\mathbf{J} = \frac{\partial z^T}{\partial \psi}$ is the $d_z \times d_\psi$ Jacobian matrix of the noise transformation.

Finally:

$$\nabla_\psi \mathcal{L}(\boldsymbol{\psi}) = \mathbb{E}_{q_0(\varepsilon)}[\nabla_\psi l(\boldsymbol{\psi}, \mathbf{z}) + \mathbf{J}^T \nabla_z l(\boldsymbol{\psi}, r(\boldsymbol{\psi}, \varepsilon)] \tag{200}$$

Delta Method

The **delta method** approximates the expectation of a function of a r.v. by the expectation of the functions taylor expansion.

If $\boldsymbol{\theta} \sim q$ with $\mathbb{E}_q(\theta)[\boldsymbol{\theta}] = \mathbf{m}$, then the **first order delta method** gives:

$$\mathbb{E}_q(\theta)[f(\boldsymbol{\theta})] \approx \mathbb{E}_q(\theta)[f(\mathbf{m}) + (\boldsymbol{\theta} - \mathbf{m})^T \nabla_\theta f(\boldsymbol{\theta})|_{\theta=m}] \approx f(\mathbf{m}) \tag{201}$$

and thus if $f = \nabla_\theta \mathcal{L}$:

$$\mathbb{E}_q(\theta)[\nabla_\theta \mathcal{L}(\boldsymbol{\theta})] \approx \nabla_\theta \mathcal{L}(\mathbf{m}) \tag{202}$$

Similarly, the **second order delta method** takes $f = \nabla^2_\theta \mathcal{L}$:

$$\mathbb{E}_q(\theta)[\nabla^2_\theta \mathcal{L}(\boldsymbol{\theta})] \approx \nabla^2_\theta \mathcal{L}(\mathbf{m}) \tag{203}$$

Gumbel softmax trick

When working with discrete variables, we cannot use the reparamterization trick. However, we can often relax the discrete variables to continuous ones in a way which allows the trick to be used.

Idea: reparamerize the categorical distribution (one hot k-ary vectors).

1. Sample $u_k \sim Unif(0, 1)$
2. Compute $\varepsilon_k = -log(-log(u_k))$
3. Project to k-simplex and relax argmax to softmax:

$$x_k = softmax(log\alpha_k + \varepsilon_k/\tau) \tag{204}$$

which smoothly approaches the discrete distribution as the **temperature** $\tau \to 0$. The result is the **gumbel-softmax distribution**. We can now take reparameterized gradients of x.

Straight-through estimator

When we quantize a signal, we replace non-differentiable maps with identity functions for the backward pass.

Alternatively, to ensure the gradients don't get too large, we can use the **hard tan function**

Bound optimization (MM) algorithms

In the context of minimization, MM stands for **majorize-minimize**. In the context of maximization, MM stands for **minorize-maximize**.

Assume our goal is to *maximize* som function $l(\boldsymbol{\theta})$ wrt $\boldsymbol{\theta}$.

The basic approach involves constructing a **surrogate function** $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t)$ which is a tight lower bound to $l(\boldsymbol{\theta})$ s.t.:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \leq l(\boldsymbol{\theta}) \tag{205}$$

$$Q(\boldsymbol{\theta}^t, \boldsymbol{\theta}^t) = l(\boldsymbol{\theta}^t) \tag{206}$$

In this case, we say **Q minorizes l**.

We perform the update:

$$\boldsymbol{\theta}^{t+1} = argmax_{\theta} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) \tag{207}$$

which guarentees a monotonic increases in the original objective.

If you don't notice monotonic improvement you have a bug in your code.

EM Algorithm

Expectation Maximization is an MM algorithm used to compute MLE or MAP estimates for probability models that have **missing data** or **hidden variables**.

In the **Expectation (E) step** we estimate the hidden variables or missing data, and then using the fully observed data, compute the MLE during the **maximization (M) step**.

The objective is to maximize the log likelihood of the observed data:

$$l(\boldsymbol{\theta}) = \sum_{n=1}^{N_D} logp(\mathbf{y}_n|\boldsymbol{\theta}) = \sum_{n=1}^{N_D} log[\sum_{z_n} p(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})] \tag{208}$$

where $\mathbf{y}_n$ are the visible variables, and $\mathbf{z}_n$ are the hidden ones.

The log sum is hard to optimize. We considr an arbitrary set of distribution $q_n(\mathbf{z}_n)$ over the hidden variables. The observed data log likelihood becomes:

$$l(\boldsymbol{\theta}) = \sum_{n=1}^{N_D} log[\sum_{z_n} q_n(\mathbf{z}_n) \frac{p(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})}{q_n(\mathbf{z}_n)}] \tag{209}$$

Log is concave so by Jensens inequality, it can be moved inside the expectation to get the following lower bound:

$$l(\boldsymbol{\theta}) \geq \sum_{N} \sum_{z_n} q_n(\mathbf{z}_n) log \frac{p(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})}{q_n(\mathbf{z}_n)} \tag{210}$$

$$= \sum_{n} \mathbb{E}_{q_n}[logp(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})] + \mathbb{H}(q_n) \tag{211}$$

$$=: \sum_{n} \mathcal{L}_{elbo}(\boldsymbol{\theta}, q_n|\mathbf{y}_n) \tag{212}$$

$$=: \mathcal{L}_{elbo}(\boldsymbol{\theta}, \{q_n\}|\mathcal{D}) \tag{213}$$

where $\mathbb{H}(q)$ is th entropy, and $\mathcal{L}_{elbo}(\boldsymbol{\theta}, \{q_n\}|\mathcal{D})$ is the **evidence lower bound, or ELBO**, since it is a lower bound on the marginal likelihood $logp(\mathbf{y}_1, ..., \mathbf{y}_n|\boldsymbol{\theta})$.

E-step

The lower bound is a sum of N terms, which have the form:

$$\mathcal{L}_{elbo}(\boldsymbol{\theta}, q_n|\mathbf{y}_n) = \sum_{z_n} q_n(\mathbf{z}_n) log \frac{p(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})}{q_n(\mathbf{z}_n)} = -D_{KL}(q_n(\mathbf{z}_n)|p(\mathbf{z}_n, |\mathbf{y}_n, \boldsymbol{\theta}) + logp(\mathbf{y}_n|\boldsymbol{\theta}) \tag{214}$$

hence we can maximize the lower bound wrt q_n by setting each one to $q_n^* = p(\mathbf{z}_n|\mathbf{y}_n, \boldsymbol{\theta})$, ensuring the lower bound is tight.

In the language of MM, we set $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^t) = \mathcal{L}_{elbo}(\boldsymbol{\theta}, \{p(\mathbf{z}_n|\mathbf{y}_n; \boldsymbol{\theta}^t)\}$ and the 2 conditions are satisfied.

When we cannot compute the posteriors $p(\boldsymbol{z}_n|\boldsymbol{y}_n, \boldsymbol{\theta}^t)$ exactly, we can approximate with a distribution $q(\mathbf{z}_n|\mathbf{y}_n; \boldsymbol{\theta}^t)$. This yields a non-tight lower bound on the log likelihood, and it called **variational EM**.

M-step

We need to maximize $mathcal L_{elbo}(\boldsymbol{\theta}, \{q_n^t\})$ wrt $\boldsymbol{\theta}$ where the distributions are the result of teh E-step at iteration t: $q_n^t = p(\mathbf{z}_n|\mathbf{y}_n; \boldsymbol{\theta}^t)$.

We can drop the entropy term (constant wrt parameters) and are left with:

$$l^t(\boldsymbol{\theta}) = \sum_n \mathbb{E}_{q_n^t(z_n)}[logp(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})] \tag{215}$$

which is the **expected complete data log likelihood**.

Thus we put:

$$\boldsymbol{\theta}^{t+1} = argmax_\theta \sum_n \mathbb{E}_{q_n^t(z_n)}[logp(\mathbf{y}_n, \mathbf{z}_n|\boldsymbol{\theta})] \tag{216}$$

This algorithm is used for example in fitting **mixture models**.

- **Generalized EM** is what we do when we can perform the E step exactly, but not the M step. In this case, we monotonically increase (instead of maximize) the expected complete data log likelihood, by performing a descent algorithm.

Bayesian Learning Rule

Recall the standard **empirical risk minimization ERM** problem, which has the form $\boldsymbol{\theta}_* = argmin_\theta \bar{l}(\boldsymbol{\theta})$ where:

$$\bar{l}(\boldsymbol{\theta}) = \sum_{n=1}^N l(\mathbf{y}_n, f_\theta(\mathbf{x}_n)) + R(\boldsymbol{\theta}) \tag{217}$$

where f_θ is prediction function, l is loss function, and R is a regularizer. Although R can prevent overrfitting, ERM can still result in parameter estimates that are not robust. A better approach is to fit a *distribution* over parameter values $q(\boldsymbol{\theta})$.

To avoid collapse to a delta function we add a KL penalty tos ome prior $\pi(\boldsymbol{\theta}) \propto exp(-R(\boldsymbol{\theta}))$ giving rise to the **bayesian learning rule objective**:

$$\mathcal{L}(q) = \mathbb{E}_{q(\theta)}[\sum_{n=1}^N l(\mathbf{y}_n, f_\theta(\mathbf{x}_n))] + D_{KL}(q_\theta|\pi(\boldsymbol{\theta})) = \mathbb{E}_{q(\theta)}[\bar{l}(\boldsymbol{\theta})] - \mathbb{H}(q(\boldsymbol{\theta})0) \tag{218}$$

- Gradient descent follows from assuming isotropic gaussian family and performing a first order delta approximation
- Newtons second order optiimzation follows from assuming a full gaussian family and applying the second order delta approximation
- Putting the log-likelihood as our loss, the bayesian learning rule optimizes for the posterior
- **Variational online gauss-newton** is a second order optimization method by assuming a diagonal gaussian approximation to the posterior, using the delta method, approximating gradient and hessian with minibatch, and optionally replace hessian with squared gradients (such as in DNN when hessian may not be positive definite)

Sometimes an objective has discrete variables (may not be differentiable). BLR uses a parametric family on the discrete set which may be smooth, and hence we can optimize.

Bayesian Optimization

Bayesian optimization is a model-based approach to black-box optimization designed for the case where the objective functino $f : \mathcal{X} \to \mathbb{R}$ is expensive to evaluate. Eg. running a simulation or hyperparameter search.

Since the true function f is expensive to evaluate we want to make as few function call (**queries to the oracle f** as possible. We build a **surrogate function** based onthe data collected so far $\mathcal{D}_n = \{(\mathbf{x}_i, y_i)|i = 1, ..., n\}$ which we can use to decide which point ot query next.

Sequential Model-Based Optimization

We alternate between querying the function at a point, and updating our estimate of the surrogate based on the new data.

At each iteration we have a labeled dataset $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : i = 1, ..., n\}$ which records points $\mathbf{x}_i$ that we have queried, and the corresponding functino values $y_i = f(\mathbf{x}_i) + \varepsilon_i$. We use the data to estimate a distribution over the true function f: $p(f|\mathcal{D}_n)$. We then choose the next point to query $\mathbf{x}_{n+1}$ using an **acquisition function** $\alpha(\mathbf{x}, \mathcal{D}_n)$ which computes the expected utility of querying $\mathbf{x}$. We again query, update beliefs, and repeat.

```
def bayesopt(f, a)
    D_0 = initialdata()
    p_0 = prior(f)

    for n = 1, ...:
        x_np1 = argmax_xa(x, D_n)
        y_np1 = f(x_np1) + eps
        D_np1 = D_n + (x_np1, y_np1)
        p_np1 = update(p, f, D_np1)
```

Surrogate Functions

How to represent and update posterior over functions $p(f|\mathcal{D}_n)$?

We can use **Gaussian process** for our surrogate. The basic idea is to represent $p(f, |\mathcal{D}_n)$ as a gaussian:

$$p(f(\mathbf{x})|\mathcal{D}_n) = \mathcal{N}(f|\mu_n(\mathbf{x}), \sigma_n^2(\mathbf{x})) \tag{219}$$

where $\mu_n(\mathbf{x}), \sigma_n(\mathbf{x})$ are functions that can be derivaed from the training data $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : i = 1, ..., n\}$.

The GP requires specifying a kernel function $\mathcal{K}_\theta(\mathbf{x}, \mathbf{x}')$ which measures similarity between input points $\mathbf{x}, \mathbf{x}'$. If the kernel matches points as similar, the corresponding funciton values are alos likely to be similar, and hence the output is positively correlated.

- GP work well when we have little training data (non parametric)
- Exact upates takes $\mathcal{O}(N^3)$ for N samples
- we need a good kernel

We can instead use a parametric model **Bayesian neural networks.**

Acquisition Functions

An **acquisition function** evaluates the expected utility of each possible point we could query:

$$\alpha(\mathbf{x}|\mathcal{D}_n) = \mathbb{E}_{p(y|x,\mathcal{D}_\backslash}[U(\mathbf{x}, y; \mathcal{D}_n)] \tag{220}$$

where U is a utility function.

- U = probability of improvement

- U = expected improvement
- U = upper confidence bound **UCB**
- u = $\mathbb{H}(x^*|\mathcal{D}_n) - \mathbb{H}(x^*|\mathcal{D}_\setminus \cup \{(x,y)\})$ **entropy search**
- **Knowledge gradient** looks ahead two steps and considers the improvement we expect to get if query x, update our posterior, and then exploit our knowledge by maximizing wrt our new beliefs.

Optimizing the acquisition function is hard and we usually use grid search or the cross entropy method with a mixture of gaussians.

Derivative Free Optimization

This is a class of techniques for optimizing functions without using derivatives. Useful for blackbox optimization, and discrete settings. If the function is expensive to evaluate, we can use Bayesian optimization. If the function is cheap to evaluate, we can use these.

Local Search

A heuristic optimization method that tries to find *global maximum* in a *discrete, unstructured search space*. The form is:

$$\mathbf{x}_{t+1} = argmax_{x \in nbr(x_t)} \mathcal{L}(\mathbf{x}) \tag{221}$$

where we maximize over neighbours. This is called **hill climbing** or **greedy search**.

Stochastic hill climbing avoid getting stuck, by placing a distribution on the neighbours that is proportional to how much they improve, and then samples one at random. Alternatively, we can run multiple greedy hill climbers in parallel from difference starting points.

In **tabu search**, we keep a memory of recently visited states, and allow exploration beyond getting stuck in local minima, so long as we have not visited such a state before.

```
    def tabu_search():
        x_0 = init()
        x_best = x_0
        while c < c_max:
            x_tp1 = argmaxf(x) for x in non-visited neighbours
            if f(x_tp1) > f(x_best)
                x_best = x_tp1
                c = 0
            else:
                c++

```

If we know knowing about the objective, we can use **random search**. This should always be tried as a baseline.

Simulated Annealing

Simulated annealing is a stochastic local search algorithm that finds the global minimum of a block box energy function by converting this energy to an unnormalized probability distribution, and then using **Metropolis Hastings** to eventually sample from one of the modes of the distribution.

Evolutionary Algorithms

Stochastic local search maintains a single best guess at each step, $\mathbf{x}_t$. If we run this for T steps with KL restarts the cost is TK. We can instead maintain a population of K good candidates $\mathcal{S}_t$ which we try to improve at each step. If we run for T steps, it still costs TK but it can get better results since it explores more of the search space in parallel. This is called an **evolutionary algorithm**.

Definition 94. The **fitness** of a member of the population is the value of the objective function. The members of the population at step $t+1$ are called the **offspring**, which are the result of randomly choosing a **parent** from $\mathcal{S}_t$, and applying a random **mutation**. Merging two parents to create offspring is called **recombination**.

The **selection function** determines how we chose parents. We can take the fittest K from the population, or the fittest K our of random samples, or via probability proportional to fitness.

The **genetic algorithm** uses **crossover mutation**, where we encode individual with binary vector, split and merge across parents to create children.

In **genetic programming**, we use a **tree representation**, which works well for optimizing over programs.

Estimation of Distribution Algorithms

We can think of population over good candidate solutions as a non-parametric density model over states with high fitness. We can learn a probabilitistic model over the configuration space that puts its mass on high scoring solutions, i.e. the population becomes the set of parameters of a generative model $\boldsymbol{\theta}_t$.

Start by creating a sample $K' > K$ candidate solutions from the current model, $\mathcal{S}_t = \{\mathbf{x}_k \sim p(\mathbf{x}|\boldsymbol{\theta}_t)\}$. Then rank the samples using the fitness function, and pick the most primising subset $\mathcal{S}_t^*$ of size K using a selection operator. Finally fit, a new probabilistic model $p(\mathbf{x}|\boldsymbol{\theta}_{t+1})$ to $\mathcal{S}_t^*$ using MLE.

This is the **estimation of distribution algorithm**.

When the population is represented by a multivariate Gaussian, we call it the **cross-entropy method.s**

Optimal Transport

Consider the families $(x_1, ..., x_n)$ and $(y_1, ..., y_n)$ of points taken from a set X. A **matching** between these two families is a bijective mapping between the sets. When matching a family with another, it is natural to consider the cost incurred when pairing any points $x_i \to y_i$. Equipped with a cost function c, the **optimal matching** is to find a permutation that reaches the smallest cost:

$$min_\sigma E(\sigma) = \sum_{i=1}^{n} c(x_i, y_{\sigma_j}) \tag{222}$$

where we optimize over permutations.

Kantorovich and Monge Discrete Formulations

The **Kantorovish** formulation allows for *mass splitting*, the idea that effort provided by one worker or needed to complete a given task can be split. To each of the n works is associated in addition to x_i a positive number $a_i > 0$ that represents the amount of time worker i is able to provide. We can introduce numbers $b_j > 0$ describing the amount of time neede to carry out each of the m tasks. So we represented workers i via the **weighted dirac measure** $a_i \delta_{x_i}$.

If we assume a balanced workload, $\sum a_i = \sum_j b_j$, the kantorovish formulation of optimal transport is:

$$OT_K(a, b) = min_{P \in \mathbb{R}_+^{n \times m}, P1_m = a, P^T 1_m = b} \langle P, C \rangle = \sum_{i,j} P_{ij} C_{ij} \tag{223}$$

Each coefficient of P_{ij} describes the allocation fo worker i to spend on task j. The ith row sum must be equal to the total a_i for the time constraint to be satisfied, and the j-th column must be equal to b_j for the output constraint.

Continuous Formulations

Let μ, υ be measures on our base measuruable space $(\mathcal{X}, \mathcal{M})$. A **push-forward** measure map is a map T such that for any measurable set $A \subset \mathcal{X}, \mu(T^{-1}(A)) = \upsilon(A)$. If T is differentiable, and μ, υ have densities p, q

w.r.t the Lebesgue measure in $\mathbb{R}^d$ this statement is equivalent due to **change of measure** to ensuring that:

$$q(T(x)) = p(x)|J_T(x)| \tag{224}$$

holds a.e., and where $|J_T(x)|$ stands for the determinant of the Jacobian matrix T evaluated at x.

In this case, we write $T_{\#}\mu = \upsilon$. The **monge formulation** finds the best map T that minimizes the average cost between x and its displacement $T(x)$:

$$inf_{T:T_{\#}\mu=\upsilon} \int_{\mathcal{X}} c(x, T(x))\mu(dx) \tag{225}$$

This is difficult to solve in practice since it's non-convex due to the pushforward constraint.

The **Kantorovish** approach also works for measures, and yields an easier linear program.

Let $\Pi(\mu, \upsilon)$ be the subset of joint probability distributions $\mathcal{P}(\mathcal{X} \times \mathcal{X})$ with marginals μ, υ.

$$\Pi(\mu, \upsilon) = \{\pi \in \mathcal{P}(\mathcal{X}^2) : \forall A \subset \mathcal{X}, \pi(\mathcal{A} \times \mathcal{X}) = \mu(A), \pi(\mathcal{X} \times A) = \upsilon(A)\} \tag{226}$$

which is non empty since it contains the product measure $\mu \times \upsilon$.

Then the **kantorovish** formulation is:

$$OT_c(\mu, \upsilon) = inf_{\pi \in \Pi(\mu,\upsilon)} \int_{\mathcal{X}}^{2} cd\pi \tag{227}$$

When c is a distance metric d exponentiated to an integer, this is called the **Wasserstein-p** distance between μ, υ:

$$W_p(\mu, \upsilon) = (inf_{\pi \in \Pi(\mu,\upsilon)} \int_{\mathcal{X}}^{2} d(x, y)^p d\pi(x, y))^{\frac{1}{p}} \tag{228}$$

Solving optimal transport

Notice that the kantarovich is problem is a linear program since the dual problem is:

$$sup_{f+g \leq c} \int_{\mathcal{X}} f d\mu + \int_{\mathcal{X}} g d\upsilon \tag{229}$$

which is a constraint and objective involving only summations.

Note that we are optimizing over two functions. If g is fixed, the optimal a must be maximal subject to the constraint, which yields the concave c-conjugate: $\hat{g}(x) := inf_y c(x, y) - g(y)$ optimal.

Since the **fenchel monroe theorem** shows conjugating twice recovers the functions, we can't simply iterate.

Instead, we keep a candidate set of $c - concave$ functions $\mathcal{F})_c = \{f : \exists g : \mathcal{X} \to \mathbb{R}, f = \hat{g}\}$

Hence we can restrict to the semi-dual formulation:

$$sup_{f \in \mathcal{F}_c} \int_{\mathcal{X}} f d\mu + \int_{\mathcal{X}} \hat{f} d\upsilon \tag{230}$$

We have removed a dual variable, narrowed the feasible set, but now have the cost of introducing the highly non-linear function $\hat{f}$. But $c - concave$ functions are nice to opimize over.

Theorem 0.95 (Kantorovish-Rubinstein duality and Lipschitz potentials)**.** *Optimizing over c-concave function, when c is a metric d (ex p=1), a c-concave function is* 1*-lipschitz and thus* $|f(x) - f(y)| \leq d(x, y), \hat{f} = -f$*. Thus:*

$$W_1(\mu, \upsilon) = sup_{f:1-lipschits} \int_{\mathcal{X}} f(d\mu - d\upsilon) \tag{231}$$

The supremum over 1-lipschitz functions can be efficiently approximated using Wavelet coefficients of densities in low dimensions, or by training neural networks parameterized to be 1-lipschitz using Relu activations, and bounds no the entries of the weight matrices.

Theorem 0.96 (Monge map as gradients of convex functions, Brenier)**.** *If $c(x,y) = \frac{1}{2}||x-y||^2$ which is up to the factor the squared W_2 distance between densities in euclidean space, the Monge map T that solves the optimization, assuming μ has a density wrt lebesgue measure, exists, and is necessarily the gradient of a convex function:*

$$T^* = argmin_{T:T_{\#}\mu=\upsilon} \int_{\mathcal{X}} \frac{1}{2}||x - T(x)||_2^2 \mu(dx) \implies T^* = \nabla u \tag{232}$$

for a convex function $u : \mathbb{R}^d \to \mathbb{R}$.

Conversely, for any convex function u, the optimal transport map between μ and $\nabla u_{\#}u$ is ∇u.

*Note: the samping complexity from using empirical measures to approximate the wasserstein distance is extremely bad. We have an exponential divergence in expectation that decreases extremely slowly in high dimensions. Therefore we always use **entropic regularization** which penalizes the kl between the coupling joint measure, and the product measure. As the weight on the regularizer drops, we approach the LP solution, but computation cost increass. The sampling complexity with regularization is good, improving at a $\mathcal{O}(\frac{1}{\sqrt{n}})$* regime.

Submodular Optimization

Let us define a *set function* $f : 2^V \to \mathbb{R}$ that assigns a value to every subset of V.

The *incremental gain of v* in the context of X is defined as $f(v|X) = f(X+v) - f(X)$ where we use addition for shorthand for union. Likewise $f(X|Y) = f(X \cup Y) - f(Y)$

A **submodular** relationship is of the form:

$$f(l|m,t) < f(l|t) \tag{233}$$

The presence of some item makes another item less valuable: the process of diminishing returns

Definition 97. A set function $f : \mathcal{P}(V) \to \mathbb{R}$ is **submodular** if for all $X, Y \subset V$:

$$f(X) + f(X) \geq f(X \cup Y) + f(X \cap Y) \tag{234}$$

Equivalently, f is **subomodular** if for all $X, Y \subset V$, where $X \subset Y$ and for all $v \notin Y$:

$$f(X+v) - f(X) \geq f(Y+v) - f(Y) \tag{235}$$

The incremental value of adding a data item X decreases as the $|X|$ grows.

Examples

- Shannon Entropy is submodular over random variables
- A function is **supermodular** iff $-f$ is submodular
- If a function is submodular and modular, it is **modular**, and corresponds to a vector-scalar pair (m, c) where $m : \mathcal{P}(V) \to \mathbb{R}$, $c \in \mathbb{R}$ and for any $A \subset V$: $m(A) = c + \sum_{v \in A} m_v$. If the modular function is normalized, $m() = 0 \implies c = 0 \implies$ the function is simply a vector.
- Non-weighted combination of submodular functions are submodular
- $f(A) = \sqrt{|A|}$ is submodular
- $f(A) = \phi(|A|)$ is subomuldar if ϕ is concave.
- A **deep submodular function** is of the form $f(A) = \phi(\sum_{u \in U} w_u \phi_u(\sum_a \in A) m_{u,a})$ where $|phi$ is an outer concave function composed with a **feature based function**, with $m_{u,a}, w_u \geq 0$

Submodular Optimization

Submodular Maximization

Due to diminishing returns, submodularity is good for diversity. Thus maximizing a submodular function requires choosign elements that are jointly dissimilar amongst each other.

Constrinat submodular optimization is NP-complete, but is very simple and efficient to approximare within 0.63 of the optimilar solution via a greedy approach.

- Start with empty set $X_0 =$
- $Xi + 1 = X_i \cup (argmax_{v \in V - X_i} f(X_i \cup \{v\})$

Which is optimal for modular functions, and approaches optimality as the **curvature** o the polymatriod function f decreases.

Submodular Minimization

This is not NP-hard, and we have polynomial algorithms for this: the *Fujishige-Wolfe.*

Inference

Basics

In probabilistic ML, all unknown quantities, predictions, hidden states, parameters are treated as r.v. and given distributions. The process of **inference** is the act of computing the posterior distribution over these quantities, conditioning on whatever data is available.

Let $\mathbf{h}$ be the unknown variables, $\mathcal{D}$ be the known variables. Given a likelihood $p(\mathcal{D}|\mathbf{h})$ and prior $p(\mathbf{h})$, we can compute the posterior $p(\mathbf{h}|\mathcal{D})$ using Bayes rule:

$$p(\mathbf{h}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{h})p(\mathbf{h})}{p(\mathcal{D})} \tag{236}$$

The computational bottleneck comes from computing the denominator, which requires solving the high dimensional integral:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{h})p(\mathbf{h})d\mathbf{h} \tag{237}$$

Common Inference Patterns

- **Global Latents**: for example parameters of a model $\boldsymbol{\theta}$ which are shared across all N observed training cases.

$$p(\mathbf{y}_{1:N}, \boldsymbol{\theta}|\mathbf{x}_{1:N}) = p(\boldsymbol{\theta})[\prod_{n=1}^{N} p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta})] \tag{238}$$

  and the goal is to compute the posterior $p(\boldsymbol{\theta}|\mathbf{x}_{1:N}, \mathbf{y}_{1:N})$

- **Local Latents**: Assume the model parameters $\boldsymbol{\theta}$ are known, the joint looks like:

$$p(\mathbf{x}_{1:N}, \mathbf{z}_{1:N}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}_x)p(\mathbf{z}_n|\boldsymbol{\theta}_z) \tag{239}$$

  and the goal is to compute $p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta})$ for each n. If the parameters are unknown, we can do EM style updates where we infer the posterior in the E step for all n simulatenously, and then update our parameters in the M step

- **Global and Local Latents**: where we model uncertainty in both the local variables $\mathbf{z}_n$ and the shared global $\boldsymbol{\theta}$ and the joint looks like:

$$p(\mathbf{x}_{1:N}, z_{1:N}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}_x)p(\boldsymbol{\theta}_z)[\prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\theta}_{\boldsymbol{x}})p(\mathbf{z}_n|\boldsymbol{\theta}_z)] \tag{240}$$

  which is being *fully bayesian*. But the uncertainty around global model parameters is much less than local latents since we have more evidence (all data vs. one point) hence we sometimes ignore.

Exact Inference Algorithms

If the prior is **conjugate** to the likelihood, the posterior will be analytically tractable, for example if prior and likelihood are exponential family.

In discrete distribution when we apply factorization (PGM), we can use DP to compute the posterior.

Approximate Inference Algorithms

- *Map Estimation*: the simplest estimate:

$$\hat{\boldsymbol{\theta}} = argmaxp(\boldsymbol{\theta}|\mathcal{D}) = argmaxlogp(\boldsymbol{\theta}) + logp(\mathcal{D}|\boldsymbol{\theta}) \tag{241}$$

and then assume that the posterior puts full probability on this single value:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \tag{242}$$

- *Grid Approximation*: partition the space into finite regions, $r_1, ..., r_k$, each representing a region of parameter space with voleum Δ centered on $\boldsymbol{\theta}_k$, then probability of begin in each region is givern by $p(\boldsymbol{\theta} \in \boldsymbol{r_k}|\mathcal{D}) \approx p_k\Delta$:

$$p_k = \frac{\hat{p}_k}{\sum_{k'=1}^{K} \hat{p}_{k'}} \tag{243}$$

$$\hat{p}_k = p(\mathcal{D}|\boldsymbol{\theta}_k)p(\boldsymbol{\theta}_k) \tag{244}$$

which fails to scale in higher dimensions

- *Laplace (Quadratic) Approximation*: where we approximate the posterior with a multivariate gaussian:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{Z}e^{-\mathcal{E}(\theta)} \tag{245}$$

where $\mathcal{E}(\boldsymbol{\theta}) = -logp(\boldsymbol{\theta}, \mathcal{D})$ is the energy function and $z = P(\mathcal{D})$ is the normalizing constant. Perform a taylor expansion around the mode $\hat{\boldsymbol{\theta}}$ (lowest energy state) we get:

$$\mathcal{E}(\boldsymbol{\theta}) \approx \mathcal{E}(\hat{\boldsymbol{\theta}}) + (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^T g + \frac{1}{2}((\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^T H(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}) \tag{246}$$

where g is the gradient at the mode, and H is the hessian at the mode. Since $\hat{\boldsymbol{\theta}}$ is the mode, the gradient is zero and thus simplifies the above. In high dim, perform diagonal approximatino of H.

- *Variational Inference*: optimization based approach to poterior inference with modeling flexibility. Approxaimte an intractable distribution $p(\boldsymbol{\theta}|\mathcal{D})$ with one that is tractable $q(\boldsymbol{\theta})$ as to minimize some discrepancy D between them:

$$q^* = argmin_{q \in \mathcal{Q}} D(q, p) \tag{247}$$

where $\mathcal{Q}$ is some tractable family of distributions. Rather than optimizing over functions, we optimize over the parameters of the functions q: called the *variational prameters* $\boldsymbol{\psi}$. Often use KL. The optimization problem reduces to:

$$\boldsymbol{\psi}^* = argmin_{\psi} D_{KL}(q(\boldsymbol{\theta}|\boldsymbol{\psi})|p(\boldsymbol{\theta}|\mathcal{D})) \tag{248}$$

$$= argmin_{\psi} D\mathbf{E}_{q(\theta)|\psi}[logq(\boldsymbol{\theta}|\boldsymbol{\psi}) - logp(\boldsymbol{\theta}) + logq(\boldsymbol{\theta}|\boldsymbol{\psi})] + logp(\mathcal{D}) \tag{249}$$

$$= argmin_{\psi} - ELBO + logp(\mathcal{D}) \tag{250}$$

Since KL is non-negative, $ELBO(\boldsymbol{\psi}) \leq logp(\mathcal{D})$ and is called the **evidence lower bound**. By maximing the elbo, we are making the variaitonal poserior closer to the true posterior. For example, we can optimize over gaussian prameters, and if we assume diagonal Σ, this is called the *mean field approximation*

- *Markov Chain Monte Carlo*: although VI is fast, it may give a biased approximation to the posterior if the specific function form is not expressive enough. A more flexible approac is to use a non-parametric approxiamtion in term of a set of sampels $q(\boldsymbol{\theta}) \approx \frac{1}{S}\sum_{s=1}^{S} \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^s)$ which is called a **monte carlo approximation**. The question is how to create posterior samples $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$ efficiently, without having to evaluate the normalizing constant.
  - For low dimensions, we can use **importance sampling**

  - For high dimensions we use MCMC, the most common form of which is called **Metropolis Hasting**
    - We start at a random point in parameter space, and then perform a random walk, by sampling new states parameters) from a *proposal distribution* $q(\boldsymbol{\theta}'|\boldsymbol{\theta})$. If we choose this carefully, the chain will visit each point in space proportionally to the posterior. We only need to compute the *density ratio*, and hence need not look at the normalizer.
    - Use a gaussian proposal $q(\boldsymbol{\theta}'|\boldsymbol{\theta}) - \mathcal{N}(\boldsymbol{\theta}'|\boldsymbol{\theta}, \sigma I)$
    - Or compute full conditionals for each variable one at a time, called **Gibbs Sampling**
    - If unknown variables are continuous, we can compute gradient of log joint $\nabla \theta logp(\boldsymbol{\theta}, \mathcal{D})$ to guide proposals to better search space, as in **Hamiltonian Monte Carlo**

- *Sequential Monte Carlo*: rather than MCMC local search, we use a sequence of different distributions, form simple to more complex with the final distribution being true posterior

Inference in State-Space Models

Definition 98. A **state space model** is a latent vairable sequence model with the conditional independencies producing a factorization of the joint like:

$$p(\mathbf{y}_{1:T}, \mathbf{z}_{1:T}|\mathbf{u}_{1:T}) = [p(z_1, u_1 \prod_{t=2}^{T} p(z_t|z_{t-1}, u_t)][\prod_{t=1}^{T} p(y_t|z_t, u_t)] \tag{251}$$

where z_t are the hidden variables at time t, y_t are the outputs (observations) and u_t are the optional inputs

Performing posterior inference about hidden states is called **state estimation**, either looking at the *filtering distribution* $p(z_t|y_{1:t})$, the *smoothin distribution* $p(z_t|y_{1:T})$ and the *fixed lag smoothing distribution* $p(z_{t-l}|y_{1:t})$. Alos the *predictive distribution h steps into the future*:

$$p(y_{t+t|y_{1:t}}) = \sum_{z_{t+h}} p(y_{t+h}|z_{t+h})p(z_{t+h}|y_{1:t}) \tag{252}$$

We are also interesting in the most probable hidden sequence $argmax_{z_{1:T}} p(z_{1:T}|y_{1:T})$, or computing samples from the posterior $z_{1:T} \sim p(z_{1:T}|y_{1:T})$

Inference Based on HMM Filter

When all the hidden variables are discrete, the SSM is known as a **hidden markov model (HMM)**, in which $p(z_1 = k) = \pi_k$ is the initial state distribution, $p(z_t = k|z_{t-1} = j) = A_{jk}$ is the state transitino matrix (assumed stationary), and $p(y_t|z_t = k)$ is the conditional distribution over observation.

Forward Filtering

The *Bayes Filter* is an algorithm for recursively computing the **belief state** $p(z_t|y_{1:t})$ given the priorr belief form the previous step $p(z_{t-1}|y_{1:t-1})$, the new observation y_t and the model, via *sequentiel Bayesian updating*.

For a dynamic model, this reduces to the *predict-update cycle*:

Predict using the *Chapman-Kolmogorov equation*:

$$p(z_t|y_{1:t-1}) = \int p(z_t|z_{t-1})p(z_{t-1}|y_{1:t-1})d_{z_{t-1}} \tag{253}$$

to compute the one step ahead predictive distribution for the latent state, updating the posterior from the previous step into the prior for the current step.

Update using *Bayes rule*

$$p(z_t|y_{1:t}) = \frac{1}{Z_t} p(y_t|z_t)p(z_t|y_{1:t-1}) = \frac{p(y_t|z_t)p(z_t|y_{1:t-1})}{p(y_t|y_{1:t-1})} \tag{254}$$

We can use the normalization constants to compute the log likelihood of the sequence:

$$logp(y_{t:T}) = \sum_{t=1}^{T} logp(y_t|y_{1:t-1}) = \sum_{t=1}^{T} logZ_t \tag{255}$$

Backwards Smoothing In the offline setting, we want to compute $p(z_t|y_{1:T})$ which is the belief about the hidden state at time t given all the data, past an future. We first perform the forward pass, and the compute smoothed belief states by working backwards. This is called **forwards filtering backwards smoothing**.

After computing $p(z_{t+1}|y_{1:T})$ by induction, we convert to a joint smothed distribution over two consecutive steps by:

$$p(z_t, z_{t+1}|y_{1:T}) = p(z_t|z_{t+1}, y_{1:T})p(z_{t+1}|y_{1:T}) \tag{256}$$

Then the new smoothed marginal is:

$$p(z_t|y_{1:T}) = \int p(z_t, z_{t+1}|y_{1:t}) \frac{p(z_{t+1}|y_{1:T})}{p(z_{t+1}|y_{1:t})} d_{z_{t+1}} \tag{257}$$

Forward Backward Algorithm

In forward pass compute:

$$\alpha_t(j) = p(z_t = j|y_{1:t}) \tag{258}$$

In backwards pass compute conditiona likelihood:

$$\beta_t(j) = p(y_{t+1:T}|z_t = j) \tag{259}$$

Combine these using:

$$\gamma_t(j_{=}p(z_t = j|y_{t+1:T,y_{1:t}}) \propto p(z_t = j, y_{t+1:T}|y_{1:t}) = \alpha_t(j)\beta_t(j) \tag{260}$$

Note: normalize after each step to avoid underflow

Note: compute log probabilities rather than probability, and use log-sum-exp for stability

Viterbi

The map estimate of one of the sequences with maximum posterior probability:

$$z^*_{1:T} = argmax_{z_{1:t}} p(z_{1:T}|y_{1:T}) \tag{261}$$

can be computed using viterbi by replacing the unnormalized forward equation sums with max:

$$\delta_t(j) = max_{z_1:t-1} p(z_{1:t-1,z_{t=j},y_{1:t}} \tag{262}$$

which can be recursively computed, by keeping track of teh most likely previous state for each possible state that we end up with.

In the backwards pass, we compute the most probable sequence using *traceback* $z^*_t = a_{t+1}(z^*_{t+1})$.

Note: viterbi can be modified to produce N best list instead of just one

Inference based on Kalman Filter

We consider state space models where all the distributions are linear Gaussian, hence:

$$p(z_t|z_{t-1}, u_t) = \mathcal{N}(z_t|F_t z_{t-1} + B_t u_t + b_t, Q_t) \tag{263}$$

$$p(y_t|z_t, u_t) = \mathcal{N}(y_t|H_t z_t + D_t u_t + d_t, R_t) \tag{264}$$

$$\tag{265}$$

Note: this is spcial case of Gaussian Bayes net, so the entire joint $p(y_{1:T}, z_{1:T}|u_{1:T}$ is a multivariate gaussian with $N_y N_z T$ dimensions.

The **Kalman filter** and **Kalman smoother** can perform exact filtering and smoothing in $O(TN_z^3)$.

Application: tracking objects from noisy measurements such as cameras or radar

In the filtering problem, we want to compute $p(z_t|y_{1:t})$. For smoothing, we want to compute $p(z_t|y_{1:T})$ using an offline dataset. The tradeoff is accuracy and delay.

Kalman Filter

The **Kalman filter** is an algorithm for exact Bayesian filtering for lienar Gaussian state space models, analogous to HMM filter.

Belief state at time t is $p(z_t|y_{1:t}) = \mathcal{N}(z_t|\mu_{t|t}, \Sigma_{t|t})$

Predict step is given by:

$$p(z_t|y_{1:t-1}, u_{1:t}) = \mathcal{N}(z_t|\mu_{t|t-1}, \Sigma_{t|t-1}) \tag{266}$$
$$\mu_{t|t-1} = F_t\mu_{t-1|t-1} + B_t u_t + b_t \tag{267}$$
$$\Sigma_{t|t-1} = F_t\Sigma_{t-1|t-1}F_t^T + Q_t \tag{268}$$

The update step is Bayes rule:

$$p(z_t|y_{1:t}, u_{1:t}) = \mathcal{N}(z_t|\mu_{t|t,\Sigma_{t|t}} \tag{269}$$
$$m_t = H_t\mu_{t|t-1} + D_t u_t + d_t \tag{270}$$
$$S_t = H_t\Sigma_{t|t-1}H_t^T + R_t \tag{271}$$
$$K_t = \Sigma_{t|t-1}H_t^T S_t^{-1} \tag{272}$$
$$e_t = y_t - m_t \tag{273}$$
$$\mu_{t|t} = \mu_{t|t-1} + K_t e_t \tag{274}$$
$$\Sigma_{t|t} = \Sigma_{t|t-1} - K_t S_t K_t^T \tag{275}$$

where m_t is the expected observation, e_t is the residual error and K_t is the Kalman gain matrix.

Posterior predictive density for the observation can be computed as:

$$p(z_t|y_{1:t-1} = \int p(z_t|z_{t-1})p(z_{t-1}|y_{1:t-1})dz_{t-1} = \mathcal{N}(\ddagger_{\sqcup}|\mu_{\sqcup|\sqcup-\infty}, \pm_{\sqcup|\sqcup-\infty} \tag{276}$$

and then marginalize out z_t:

$$p(y_t|y_{1:t-1}) = \int p(y_t, z_t|y_{1:t-1})dz_t = \mathcal{N}(y_t|m_t, S_t) \tag{277}$$

which we can generalize the prediction to K steps into the future by forecasting K steps in latent space, and then grounding the final state into predicted observations.

Kalman Smoother

The RTS smoothing sequentially computes $p(z_t|y_{1:t})$ in an offline setting, for each t.

Inference for Non Linear SSM

For low dimensional latents, discretize and use HMM filter. Otherwise approximate with mixture of gaussians and run kalman.

Inference via Local Linearization

Linearize the dynamics and obsrevations models about the previous state estimate using first order Taylor expansion, and then apply standard kalman filter. This is called the **Extended Kalman filter**.

Inference for Graphical Models

Goal: leverage the conditional independence properties encoded in graph structures to perform efficient inference, implemented via dynamic programming.

Belief Propogation on Trees

Consider a pairwise undirected graphical model, which can be written as:

$$p^*(z) = p(z|y) \propto \prod_{s \in V} \psi_s(z_s|y_s) \prod_{(s,t) \in E} \psi_{s,t}(z_s, z_t) \tag{278}$$

where $\psi_{s,t}(z_s, z_t)$ are the pairwise clique potentials, and $\psi_s(z_s|y_s)$ are the local evidence potentials.

We can define the potentials as unnormalized marginals, so that:

$$p^*(z) \propto \prod_{s \in V} p^*(z_s) \prod_{(s,t) \in E} \frac{p^*(z_s, z_t)}{p^*(z_s)p^*(z_t)} \tag{279}$$

Sum Product Algorithm

Assume the model is an undirected tree, pick a root, and orient all edges away from the root, so each node has a unique parent.

We let $m_{s \to t}(z_t)$ denote the message from node s to node t. We update the belief state of node s by combining the incoming messages from all the children with its own evidence:

$$bel_s(z_s) \propto \psi_s(z_s) \prod_{t \in ch(s)} m_{t \to s}(z_s) \tag{280}$$

To compute the outgoing message that s should send to parent t, we pass the local belief through the pairwise potential and then marginalize out s:

$$m_{s \to t}(z_t) = \sum_{z_s} \psi_{s,t}(z_s, z_t) bel_s(z_s) \tag{281}$$

At the root, $bel_t(z_t) = p(z_t|y)$ will have seen all the evidence.

It can then send messages back down the leaves. The message that s sends to its child t is:

$$m_{s \to t}(z_t) = \sum_{z_t} ()\psi_s 9z_s)\psi_{s,t}(z_s, z_t) \prod_{u \in ch(s) - t} m_{u \to s}(z_s) \tag{282}$$

and then:

$$bel_t(z_t) \propto bel_t(z_t) m_{s \to t}(z_t) \tag{283}$$

Giving using the **posterior marginals**.

Max product

We can replace the sum operation with max operation to get **max product belief propogation**, yielding the **max marginals**:

$$\zeta_i(k) = max_{z_{-i}} p(z_i = k, z_{-i}|y) \tag{284}$$

We can compute the **max of the posterior marginal MPM**: $\hat{z}_i = argmax_k \gamma_i(k)$

We can compute the **maximizer of the max marginal MMM**: $\tilde{z}_i = argmax_k \zeta_i(k)$

Loopy Belief Propagation

We can extend belief propagation to work on graphs with cycles or loops, called **Loopy belief propagation**

We don't have guaranteed convergence, but it may look good.

Loopy BP For Pairwise Undirected Graphs

Initialize all messages to the 1 vector. Then in parallel, each node absorbs messages from all its neighbours:

$$bel_s(z_s) \propto \psi_s(z_s) \prod_{t \in nbr(s)} m_{t \to s}(z_s) \tag{285}$$

Then in parallel, each node sends messages to each neighbour:

$$m_{s \to t}(z_t) = \sum_{z_s} (\psi_s(z_s) \psi_{s,t}(z_s, z_t) \prod_{u \in nbr(s)-t} m_{u \to s}(z_s)) \tag{286}$$

Keep doing this until convergence, which is guarenteed for a tree after D iteration where D is the *diameter of the graph.*

*Note: to increase chance of convergence, use **damping**, which computes linear combination of previous message with new message in an update, using a damping factor* $0 \leq \lambda \leq 1$

Application: **Affinitity Propogation**, an improvement to K-medoids clustering, taking as inpute a pairwise similarity matrix.

Let $c_i \in \{1, ..., N\}$ represent the centroid for datapoint i. The goal is to maximize:

$$J(c) = \sum_{i=1}^{N} S(i, c_i) + \sum_{k=1}^{N} \delta_k(c) \tag{287}$$

where $S(i, c_k)$ is the similarity between data point i, and its centroid c_i), and δ_k is the penalty term:

$$\delta_k(c) = \begin{cases} -\infty & if c_k \neq k \text{ but there exists } i \text{ such that } c_i = k \\ 0 \textbf{ otherwise} \end{cases} \tag{288}$$

which incentivies representative samples to vote for themselves as centroids, thus encouraging clustering behaviour.

We can find a strong local max of the objective by using max product loopy bp.

0.0.1 Variable Elimination

Goal: exactly compute posterior marginal $p(z_Q|y)$ for any query set in a graphical model p.

Idea: write out the full marginalization, use conditional independence to write the factors, pull factors into the sum as much as possible.

The **elimination order** seeks to minimizes the size of the intermediate factors that are created, but finding the optimal is np-complete. Often, we use a greedy *min-fill heuristic.*

Complexity: $O(NK^{w+1})$

Variational Inference

Variational inference reduces posterior inference to an optimization problem.

Consider a model with uknown variables z, known variables x and fixed parameters θ. Since computing the true posterior $p_\theta(z|x)$ is assumed intractable, we will use an approximation $q(z)$ which we choose to minimize the following loss:

$$q = argmin_{q \in Q} D_{KL}(q(z)|p_\theta(z|x)) \tag{289}$$

minimizing over functional family Q. In practice, we pick a parametric family Q where we use ψ to represent the **variational parameters**. We compute the best varaitional parameters (for given x) as:

$$\begin{aligned}
\psi^* &= argmin_\psi D_{KL}(q(z|\psi)|p_\theta(z|x)) && (290)\\
&= argmin \mathbb{E}_{q(z|\psi)}[logq(z|\psi) - log(\frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)})] && (291)\\
&= argmin \mathbb{E}_{q(z|\psi)}[logq(z|\psi) - logp_\theta(x|z) - logp_\theta(z)] + logp_\theta(x) && (292)\\
&= argmin_\psi \mathcal{L}(\psi|\theta, x) + logp_\theta(x) && (293)
\end{aligned}$$

The final term $logp_\theta(x) = \int p_\theta(x, z)dz$ is generally intractable, but we can drop it since its independent of ψ.

This we minimize:

$$\mathcal{L}(\psi|\theta, x) = D_{KL}(q(z|\psi)|p_\theta(x, z)) := \mathbb{E}_{q(z|\psi)}[logq(z|\psi) - logp_\theta(x, z)] \tag{294}$$

If we define $\mathcal{E}(z) = -logp_\theta(z, x)$ as the energy then we have:

$$\mathcal{L}(\psi|\theta, x) = \mathbb{E}_{q(z|\psi)}[\mathcal{E}(z)] - \mathbb{H}(q) \tag{295}$$

which is known as the **variational free energy**.

The variational free energy is the expected energy minus the entropy. So we try to minimize the expected energy with maximize entropy.

The negative of the varational free energy is called the **evidence lower bound** or **ELBO**:

$$ELBO(\psi|\theta, x) = \mathbb{E}_{q(z|\psi)}[logp_\theta(x, z) - logq(z|\psi)] \tag{296}$$

This is the **eveidence lower bound** since it lower bounds the evidence:

$$ELBO(\psi|\theta, x) \leq logp_\theta(x) \tag{297}$$

Therefore maximize the ELBO wrt ψ will decrease the original KL, since $logp_\theta(x)$ is constant wrt ψ.

We can rewrite the elbo as:

$$ELBO(\psi|\theta, x) = \mathbb{E}_{q(z|\psi)}[logp_\theta(x, z)] + \mathbb{H}(q(z|\psi)) \tag{298}$$

and so *the ELBO is the expected log joint + entorpy of the posterior*, hence the first terms encourages it to be a joint MAP configuration, while the seocnd encourages a posterior with maximum entropy.

Also:

$$ELBO = \mathbb{E}_{q(z|\psi)}[logp_\theta(x|z)] - D_{KL}(q(z|\psi)|p_\theta(z)) \tag{299}$$

so we are *maximizing the expected log likelihood, while keeping the posterior close to the prior*

Mean Field Variational Inference

A common approximation in variational inference is to assume that all the latent variables are independent:

$$q(z|\psi) = \prod_{j=1}^{J} q_j(z_j) \tag{300}$$

where J is the number of hidden variables, $q_j(z_j) = q_{\psi_j}(z_j)$ and ψ_j are the variaitonal parameters for the jth distribution.

This is called the **mean field approximation**.

In this approximation, the elbo becomes:

$$ELBO(\psi) = \int q(z|\psi) log p_\theta(x,z) dz + \sum_{j=1}^{J} \mathbb{H}(q_j) \tag{301}$$

since the entorpy of a product distirbution is the sum of entropies of each component. We either directly optimize this or do it block-coordinate wise.

0.0.2 Variational Bayes

In the bayesian setting, we treat the parameters θ as latent variables, and our goal is to compute the parameter posterior $p(\theta|\mathcal{D}) \propto p(\theta)p(\mathcal{D}|\theta)$. Applying Variational inference is to this problem is called **variational bayes**

Fixed Form Variational Inference

In mean field, we assuem that the approximating distribution $q(z)$ factorizes across variables, and did not specify the form of q_j, hence we call this "free-form" VI.

Instead, we can pick any convenient form we like for $q(z)$, such as a multivariate gaussian, and then we directly maximize the ELBO using gradient ascent. This is called **fixed-form Variational inference**.

Recall we want to maximize the lower bound:

$$ELBO(\psi|\mathcal{D}) = \mathbb{E}_{q(z|\psi)}[log \frac{p(z)p(\mathcal{D}|z}{q_\psi(z)}] = \mathbb{E}_{q_\psi}[l_\psi(z)] \tag{302}$$

wrt ψ where $l_\psi(z) = logp(z,\mathcal{D}) - logq_\psi(z)$

Stochastic Varational Inference

In many models, the likelihood factorized into a product of terms, and hence:

$$l_\psi(z) = [\sum_{n=1}^{N} logp(x_n|z)] + logp(z) - logq_\psi(z) \tag{303}$$

If N is large, we can compute an unbiased minibatch approximation to the expression as:

$$\hat{l}_\psi(z) = [\frac{N}{B} \sum_{n=1}^{B} logp(x_b|z)] + logp(z) - logq_\psi(z) \tag{304}$$

leading to an unbiased monte carlo approximation to the ELBO.

This is called **stochastic variationla inference** and allows VI to scale to large datasets.

Blackbox Variational Inference

Assume we can evaluate $l_\psi(z)$ pointwise, but we don't need to necessarily assume we can take gradeints of this function. To estimate the gradient of the ELBO, we will use the *score function estimator (REINFORCE)*. Then the gradient of the ELBO is:

$$\nabla_\psi ELBO(\psi) = \nabla_\psi \int q_\psi(z) log \frac{p(z, \mathcal{D}}{q_\psi(z)} dx \tag{305}$$

$$= \mathbb{E}_{q_\psi(z)}[\nabla_\psi log q_\psi(z) l_\psi(z)] \tag{306}$$

We can compute a stochastic approximation to this gradient by sampling $z_s \sim q_\psi(z)$ and then computing:

$$\nabla_\psi \hat{ELBO}_\psi = \frac{1}{S} \sum_{s=1}^{S} \nabla_\psi log q_\psi(z_s) l_\psi(z_s) \tag{307}$$

In paractice, the variance of this estimator is large, so it is important to use **control variates** to stabilize, a good baseline is the score function $b_i 9z) = \nabla_{\psi_i} log q_{\psi_i}(z)$ since the expected value of the score function we know is zero and is correlated with our gradient.

We can stop the algorithm when the lower bound stops increasing, using a running average over the last w observations to smooth the noise.

Reparamterization variational inference

We can exploit the **reparameterization trick** to get a lower variance estimator for the gradient. This assumes that $l_\psi(z)$ is differentiable in z, and that we can sample $z \sim q_\psi(z)$ by first sampling noise $\varepsilon \sim q_0(\varepsilon)$ and hten trasnforming it to compute the latent random vairables $z = r(\psi, \varepsilon)$. In this case the ELBO becomes:

$$ELBO(\psi) = \mathbb{E}_{q_0(\varepsilon)}[l_\psi(r(\psi, \varepsilon))] = \mathbb{E}_{q_0(\varepsilon)}[log p(r(\psi, \varepsilon), \mathcal{D}) - log q_\psi(r(\psi, \varepsilon))] \tag{308}$$

Since the sampling distribution $q_0(\varepsilon)$ is independent of the variational parameters ψ we can push the gradient operator inside the expectation, adn thus we can estiamte the gradient using ato-diff.

Often we approxiamate with a family of gaussian, either full rank diagonal, or isotopic.

Automatic Differentation Variational Inference To apply gaussian VI, we need to transform constrained prameters (such as variance terms) to unconstrained form so they live in $\mathbb{R}^D$. This is called **auto differentiation variational inference**.

We need to be able to form a bijection from the distribution to euclidean space.

For example, we can use block-factored Gaussian variaitonl approximation to the posterior:

$$q(z|\psi) = \prod b = 1^B \mathcal{N}(z_b|\mu_b, \Sigma_b) \tag{309}$$

where $z = T(\theta)$ is a bijective mapping that maps the constrained space to unconstrained space.

By the change of variable formula:

$$p(z) = p(T^{-1}(z)) |det(J_{T^{-1}}(z))| \tag{310}$$

where $J_{T^{-1}}$ is the Jacobian of the inverse $z \to \theta$. The ELBO becomes:

$$ELBO(\psi) = \mathbb{E}_{z \sim q(z|\psi)}[p(\mathcal{D}|T^{-1}(z)) + log p(T^{-1}(z)) + log|det(J_{T^{-1}(z)}(z)|] + \mathbb{H}(\psi) \tag{311}$$

which we can use monte carlo approximation over z for, along with the reparameterization trick: $z = \mu + \sigma \cdot \varepsilon, \varepsilon \sim \mathcal{N}(O, I)$.

Amortized Inference

Suppose we want to perform parameter estimation in a model with local latents:

$$\hat{\theta} = argmax_\theta \sum_{n=1}^{N} log \sum_{z_n} p_\theta(x_n, z_n) \tag{312}$$

When using the above algorithms, we therefore have to solve an optimization problem for each posterior $p_\theta(z_n|x_n)$ for each example n, which is slow.

An alternative approaxh is to train a model known as an **inference network** to predict ψ_n from the observed data x_n. This is called **amortized inference** since we are reducing the per-example time inference by training a model that is shared across all examples.

Then:

$$q(z_n|\psi_n) = q(z_n|f_\phi^{inf}(x_n)) = q_\phi(z_n|x_n) \tag{313}$$

The **amortized elbo** becomes:

$$ELBO(\phi, \theta|\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N}[\mathbb{E}_{q_\phi(z_n|x_n)}[logp_\theta(x_n, z_n) - logq_\phi(z|x_n)]] \tag{314}$$

which we can approximate by sampling a single data point $x_n \sim p_\mathcal{D}$ and then sampling a single latent $z_n \sim q_\phi(z_n|x_n)$. If the posteriors are reparameterizable, we can push gradients inside and apply SGD.

Improvements

We can improve our inference by optimizing over more general family of distributions.

We can also leverage structured inference techniques in conjuction with variational approximations.

We can also optimize a tighter bound on the evidence by using importance weighting.

Importance weighted autoencoder bound

Let $q_\phi(z|x)$ be the inference network, viewed as a proposal distribution fro the target posterior $p_\theta(z|x)$. Let $w_s^* = \frac{p_\theta(x,z_s)}{q_\phi(z_s|z)}$ be the unnormalized importance weight for a sample, and $w_s = w_s^*/(\sum_{s'} w_{s'}^*)$ be the normalized importance weight.

We can compute an estimate of the marginal likelihood $p(x)$ using:

$$\hat{p}_S(x|_{z_{1:S}} = \frac{1}{S}\sum_{k=1}^{S} w_s \tag{315}$$

which is unbiased. Since the estimator is always positive, we take logirthms and obtain a stochastic lower bound on the log likelihood:

$$ELBO(\phi, \theta|x) = \mathbb{E}_{q_\phi(z_{1:s}|x)}[log\hat{p}_S(z_{1:s}] \leq log\mathbb{E}_{q_\phi(z_{1:s}|_x}[\hat{p}_s(z_{1:s})] = logp(x) \tag{316}$$

via Jensens inequality. This bound gets tighter with more samples S.

Wake-sleep algorithm

So far, we fit latent variable models by maximing ELBO. First, this doesn't work with discrete latents because we cannot use the raparameterization trick and so we have to use higher variance estimators such as REINFORCE. Second, the bound may not be tight.

A different way to jointly train generative and inference models is called the **wake-sleep** algorithm. In the wake phase, we optimize the generative model parameters θ to maximize the marginal likelihood of the observed data. In the sleep phase, we optimize the inference model parameters ϕ to invert the generative model by training the inference network on labeled (x, z) pairs, where x are samples generated by the current model parameter's.

Expectation propagation

One problem with lower bound maximization, is the we are minimizing a KL. This means that $q(z|x)$ tends to be too compact (over-confident) to avoid the situation $q(z|x) > 0, p(z|x) = 0$ which would incur infinite KL.

This might be fine for multi-modal posteriors (mixture models), but not for unimodal posterios (bayesian logistic regression, Gaussian processes with log-concave likelihoods)

Expectation propogation avoids this, by performing a local approxiamtion to $D_{KL}(p|q)$.

recall: $D_{KL}(p|q)$ results in broad posteriors

Assume the exact posterior can be written:

$$p(\theta|\mathcal{D}) = \frac{1}{Z_p}\hat{p}(\theta), \hat{p}(\theta) = p_0(\theta)\prod_{k=1}^{K} f_k(\theta) \tag{317}$$

wherre p_0 is the prior, and f_k is the kth likelihood term or **local factor**. We approximate the postrior:

$$q(\theta) = p_0(\theta)\prod_{k=1}^{K} \hat{f}_k(\theta) \tag{318}$$

where $\hat{f}_k \in \mathcal{Q}$ is the approximate local factor in a tractable family, usually gaussian or exponential family.

We optimize each $\hat{f}_i$ in turn, keeping others fixed, and then projection to the model family via the KL.

Monte Carlo Inference

Monte carlo methods are a stochastic approach to solving numerical integration problems.

We often want to compute the exepcted value of some function of a random variable $\mathbb{E}[f(X)]$. This requires computing:

$$E[f(x)] = \int f(x)p(x)dx \tag{319}$$

where $p(x)$ is the target distribution of X. In low dimensions (N¡=3) we can compute the above efficiently using **numerical integration** which adaptively computes a grid, and then evalutes the function at each point on the grid. This does not scale by curse of dimensionality.

An alternative approach is to draw multiple random samples $x_n \sim p(x)$ and then compute:

$$\mathbb{E}[f(x)] \sim \frac{1}{N_s}\sum_{n=1}^{N_s} f(x_n) \tag{320}$$

This is called **monte carlo integration.** The advantage over numerical integration is that the function is only evaluated in places where there is non-negligible probability so it does not need to uniformly cover the space. The catch is we need a way to generate samples, and that it may have high variance.

The estimate is unbiased, and by the central limit theorem, the variance decreases linearly with number of samples.

The quantity $\sqrt{\frac{\hat{\sigma}^2}{N_s}}$ is called the **empirical standard error** and is an estimate of our uncertainty about our estimate of μ. The nice part is this is independent of dimensional!

Generating Random samples from simple distributions

The simplest method for sampling from a uni-variate distribution is based on the **inverse probability transform**. Let F be a cdf of some distribution we want to sample from.

Theorem 0.99. *If $U \sim U(0,1)$ is a uniform rv, then $F^{-1}(U) \sim F$*

hence we can sample from any universriate distribution for which we can evaluate its inverse cdf.

Sampling from a gaussian (Box-Muller Method)

Idea:sample uniformly from a unit circle and then use change of variables to derive samples from a spherical 2d gaussian.

Sample $z_1, z_2 \in (-1, 1)$ uniformly, and discard pairs that do not live in the circle. The points will be uniformly distributed in the circle.

Let $x_i = z_i(\frac{-2lnr^2}{r^2})^{\frac{1}{2}}$, where $r^2 = z_1^2 + z_2^2$ and using multivariate change of variables:

$$p(x_1, x_2) = p(z_1, z_2)|\frac{\partial(z_1, z_2)}{\partial(x_1, x_2)}| = [\frac{1}{\sqrt{2\pi}}exp(-\frac{1}{2}x_1^2)][\frac{1}{\sqrt{2\pi}}exp(-\frac{1}{2}x_2^2)] \tag{321}$$

hence x_1, x_2 are two independent samples from a uni-variate Gaussian.

To sample form multivariate gaussian, we first compute Cholesky decomposition of the covariance $\Sigma = LL^T$, we sample $x \sim N(0, I)$ using Box-muller, ans the set $y = Lx + \mu$.

Rejection Sampling

Suppose we want to sample from a target distribution $p(x) = \hat{p}(x)/Z_p$

In **rejection sampling** we require acces to a *proposal distribution* $q(x)$ that satisfies:

$$Cq(x) \geq \hat{p}(x) \tag{322}$$

for some constant C. The function $Cq(x)$ provides an upper envelope for $\hat{p}$.

First sample $x_0 \sim q(x)$ which corresponds to picking a random x location.

Then sample $u_0 \sim Unif(0, Cq(x_0))$ which corresponds to picking a random heigh under the envelope.

If $u_0 > \hat{p}(x_0)$ we reject the sample, otherwise we accept it.

The acceptance probability is given by $\frac{1}{C}$, so we want to choose a small C while satisfying the envelope inequality $Cq(x) \geq \hat{p}(x)$.

Note: in high dimensions, the acceptance rate often decays exponentially fast with dimension

Importance Sampling

Goal of **importance sampling** is to approximate integrals of the form:

$$\mathbb{E}[\phi(x)] = \int \phi(x)\pi(x)dx \tag{323}$$

where ϕ is called a *target function*, $\pi(x)$ is the *target distirbution* which is often a conditional distribution of the form $p(x|y)$. Since it is difficult to draw from the target distribution, we will instead draw form some **proposal distribution** $q(x)$. We then adjust for the innaccuracies of this by associating weights with each sample, so we end up with a weighted monte carlo approximation.

$$\mathbb{E}[\phi(x)] \approx \sum_{n=1}^{N} W_n \phi(x_n) \tag{324}$$

In **direct importance sampling** we assume that we can *evaluate* the normalized target distribution $\pi(x)$, but we cannot sample from it

$$\int \phi(x)\pi(x)dx = \int \phi(x)\frac{\pi(x)}{q(x)}q(x)dx \tag{325}$$

Note: we require that the proposal be non-zero whenever the target is non-zero. i.e. trhe support of q is greater than or equal to the support of π

Then en unbiased estimate of the true mean $\mathbb{E}[\phi(x)]$ is:

$$\mathbb{E}[\phi(x)] \approx \frac{1}{N_s}\sum_{n=1}^{N_s}\frac{\pi(x_n)}{q(x_n)}\phi(x_n) = \frac{1}{N_s}\sum_{n=1}^{N}\hat{w}_n\phi(x_n) \tag{326}$$

where $\hat{w}_n = \frac{\pi(x_n)}{q(x_n)}$

The disadvantage of direct importance samlping is that we need a way to evaluate the normalized target distirbution π in order to compute weights. It is often easier to evalute the *unnormalized target distribution*: $\gamma(x) = Z\pi(x)$.

The key idea of **self-normalized importance sampling** is to approximate the normalization constant Z with importance sampling.

The resulting estimate is a ratio of two estimates, and hece is biased, however as $N_s \to \infty$ bias goes to zero.

We use:

$$\mathbb{E}[\phi(x)] = \int \phi(x)\pi(x)dx = \frac{\int \phi(x)\gamma(x)dx}{\int \gamma(x)dx} = \frac{\int [\frac{\gamma(x)}{q(x)}\phi(x)]q(x)dx}{\int [\frac{\gamma(x)}{q(x)}]q(x)dx} \approx \frac{\frac{1}{N_s}\sum_{n=1}^{N_s}\hat{w_n}\phi(x_n)}{\frac{1}{N_s}\sum_{n=1}^{N_s}\hat{w}_n} \tag{327}$$

where $\hat{w}_n = \frac{\gamma(x_n)}{q(x_n)}$

Thus:

$$\mathbb{E}[\phi(x)] \approx \sum_{n=1}^{N_s} W_n \phi(x_n) \tag{328}$$

with

$$W_n = \frac{\hat{w}_n}{\sum_{n'=1}^{N_s} \hat{w}_{n'}} \tag{329}$$

This is equivalent to approximating the target distribution using a weighted sum of delta functions:

$$\pi(x) \approx \sum_{n=1}^{N_s} W_n \delta(x - x_n) \tag{330}$$

The performance of importance sampling depends crucially on the quality of the proposal distribution. At the minimum, the support of q needs to contain the support of the target.

One way to come up with a good proposal is to learn one by optimizing the variational lower bound or ELBO.

Controlling Monte Carlo variance

The standard error in a monte carlo estimate is $O(\frac{1}{\sqrt{S}})$ where S is the number of independent sampes.

Common Random numbers

The *common random numbers trick* involves using the same random samples z_s for evaluting to expectations, so that the differnece is attributed to the parameters and not noise.

Rao-Blackwellisation

Suppose we have to rv X, Y and we want to estimate $\overline{f} = \mathbb{E}[f(X, Y)]$. The naive MC approximation is:

$$\hat{f}_{MC} = \frac{1}{S} \sum_{s=1}^{S} f(X_s, Y_s) \tag{331}$$

where $(X_s, Y_s) \sim p(X, Y)$.

Suppose we can analytically marginalize out Y, provided we know X: so we can tractably comptue:

$$f_X(X_s) = \int dY p(Y|X_s) f(X_s, Y) = \mathbb{E}[f(X, Y)|X = X_s] \tag{332}$$

The **Rao-Blackwellised estimator** is:

$$\hat{f}_{RB} = \frac{1}{S} \sum_{s=1}^{S} f_X(X_s) \tag{333}$$

where $X_s \sim p(X)$. This is still unbiased, but has lower variance since we are sampling in a reduced dimensional space.

Control Variates

Suppose we want to estimate $\mu = \mathbb{E}[f(X)]$ using an unbiased estimator $m(X) = \frac{1}{S} \sum_{s=1}^{S} m(x_s)$ where $x_s \sim p$ and $\mathbb{E}[m(X)] = \mu$.

Consider the alternative estimator:

$$m^*(X) = m(X) + c(b(X) - \mathbb{E}[b(X)]) \tag{334}$$

this is called a **control variate**, and b is called a **baseline**.

This is obivously unbiased, and if b is correlated with m it has lower variance since:

$$\mathbb{V}[m^*(X)] = \mathbb{V}[m(X)] + c^2\mathbb{V}[b(X)] + 2cCOV[m(X), b(X)] \tag{335}$$

We can find optimal value of c by differentiating and see that:

$$c^* = -\frac{COV[m(X), b(X)]}{\mathbb{V}[b(X)]} \tag{336}$$

and hence the corresponding variance is:

$$\mathbb{V}[m^*(X)] = (1 - \rho^2_{m,b})\mathbb{V}[m(X)] \leq \mathbb{V}[m(X)] \tag{337}$$

so long as the correlation $\rho_{m,b}$ is high.

Antithetic Sampling

Suppose we want to estimate $\theta = \mathbb{E}[Y]$ and let Y_1, Y_2 be two samples. An unbiased estimate of θ is $\hat{\theta} = \frac{Y_1+Y_2}{2}$ with variance:

$$\mathbb{V}[\hat{\theta}] = \frac{\mathbb{V}[Y_1] + \mathbb{V}[Y_2] + 2COV[Y_1, Y_2]}{4} \tag{338}$$

so the variance is reduced if $COV[Y_1, Y_2] < 0$.

So whenever we sample Y_1 we should set Y_2 to be its opposite, but with the same men

Markov Chain Monte Carlo (MCMC) Inference

Non-iterative Monte carlo methods like rejection sampling and importance sampling generate independent samples from a target distribution, but do not work well in high dimensional spaces.

The basic idea of **Markov chain monte carlo** is to construct a markov chain on the state space $\mathcal{X}$ whose stationary distribution is the target density $p^*(x)$ of interest.

We perform a random walk on the state space in such a way that the fraction of tiem we spend in each state x is proportional to $p^*(x)$. BY drawing (correlated) samples $x_0, x_1, ...$ from the chain, we can perform monte carlo integration wrt p^*.

The initial samples from the chain do not come from the stationary distribution, and should be discarded. The amount of time it takes to reach statinoarity is called the **burn in time**, and reducing this makes the algorithm fast.

Metropolis Hastings Algorithm

The basic idea of metropolis hastings is that at each step, we propose to move from the current state x to a new state x' with probability $q(x'|x)$ where q is called the **proposal distribution** or **kernel**.

The user if basically free to use any kind of proposal they want.

Having proposed a move to x', we then decide whether to **accept** this proposal or reject it, which we do such that the long term fration of time spent in each stte is proportional to p^*.

If the proposal is accepted we move to new state x' else we stay in x.

If the proposal is symmetric: $q(x'|x) = q(x|x')$ then the acceptance probability is given by:

$$A = min(1, \frac{p^*(x')}{p^*(x)} \tag{339}$$

If the proposal is asymmetric, so $q(x'|x) \neq q(x|x')$ we need the **Hastings correction** given by:

$$A = min(1, \alpha) \tag{340}$$

$$\alpha = \frac{p^*(x')q(x|x')}{p^*(x)q(x'|x)} \tag{341}$$

which compensates for the fact that the proposal distribution itself (rather than just the target distribution) might favor certain states.

An important reaons why Metropolis Hasting is useful is that when evaluating α, we only need to know the target density up to a normalization constant.

Theorem 0.100. *If the transition matrix defined by the Metropolis Hasting algorithm is ergodic and irreducible then p^* is its unique limiting distribution.*

Proposal Distributions

We can use an **independence sampler** s.t. $q(x'|x) = q(x')$ where the new state is independent of the old state. We often use gaussian since it has nonzero density on the entire state space, and hence is a valid proposal for any unconstrained continous state space.

The **random walk Metropolis** algorithm corresponds to Metropolish Hasting with the proposal:

$$q(x'|x) = \mathcal{N}(x'|x, \tau^2 I) \tag{342}$$

Here τ is a scale factor chosen to facilitate rapid mixing.

If there are several proposals that might be useful one can combine them using a **mixture proposal** which is a convex combination of base proposals.

In the case where the target distribution is a posterior $p^*(x) = p(x|\mathcal{D})$ it is helpful to condition the proposal not just on the previous hidden state but also the data so that $q(x'|x, \mathcal{D})$ which is called **data-driven MCMC**. We can train a recognition network to propose states using $q(x'|x, \mathcal{D}) = f(x)$.

Initialization

We need to start MCMC in an initial state with non-zero probability. A natural approach is to first use an optimizer to find a local mode, or close to one.

Gibbs Sampling

The major problems with Metropolis Hastings are the need to chose the proposal distirbution, and the fact that the acceptance rate may be low.

Gibbs sampling is a method that exploits conditional independence properties of a graphical model to automatically create a good proposal with acceptance probability 1.

The idea is to sample from each variable in turn, conditioned on the values of all other variables in the distirbution.

For example, if D=3 variables:

$$x_1^{s+1} \sim p(x_1|x_2^s, x_3^s) \tag{343}$$
$$x_2^{s+1} \sim p(x_2|x_1^{s+1}, x_3^s) \tag{344}$$
$$x_3^{s+1} \sim p(x_3|x_1^{s+1}, x_2^{s+1} \tag{345}$$

Note: Gibbs sampling is a special case of metropolis hastings where we use a sequence of proposals of the form:

$$q_i(x'|x) = p(x_i'|x_{-i})\mathbb{I}[x_{-i}' = x_{-i}] \tag{346}$$

Gibbs sampling has an acceptance rate of 100%, but only updates one coordinate at a time. We can exploit conditional independence to sample multiple variables in parallel. For example, in a 2d pixel grid, we can checkboard pattern the pixels, and sample black nodes in parallel, then white nodes in parallel.

Hamiltonian Monte Carlo

Hamiltornian Monte Carlo leverages gradient information to guide local moves.

Consider a particle rolling around an energy landscape. We can characterize the motion of the particle in terms of its position $\theta \in \mathbb{R}^D$ and its momentum $v \in \mathbb{R}^D$. The set of possible values (θ, v) is called the **phase space.**

The **Hamiltonian function** for each point in phase space is:

$$\mathcal{H}(\theta, v) = \mathcal{E}(\theta) + \mathcal{K}(v) \tag{347}$$

where $\mathcal{E}(\theta)$ is the **potential energy** and $\mathcal{K}(v)$ is the **kinetic energy**, and the **hamiltonian** is the total energy.

We often take the potential energy to be:

$$\mathcal{E}(\theta) = -log\hat{p}(\theta) \tag{348}$$

where $\hat{p}(\theta)$ is a possible unnormalized distribution such as $p(\theta, \mathcal{D})$ and the kinetic energy to be:

$$\mathcal{K}(v) = \frac{1}{2} v^T \Sigma^{-1} v \tag{349}$$

where Σ is a positive definite matrix known as the **inverse mass matrix**.

The trajectory of a particle within an energy level set can be obtained by solving to continuous time differential equation sknown as **Hamiltons Equations**:

$$\frac{d\theta}{dt} = \frac{\partial \mathcal{H}}{\partial v} = \frac{\partial \mathcal{K}}{\partial v} \tag{350}$$

$$\frac{dv}{dt} = -\frac{\partial \mathcal{H}}{\partial \theta} = -\frac{\partial \mathcal{E}}{\partial \theta} \tag{351}$$

Note: the mapping $(\theta(t), v(t)) \rightarrow (\theta(t+s), v(t+s))$ for s small is invertible and volume preserving (Jacobian 1)

Integrating Hamiltons Equations

The simplest way to model the time evolution is to update the position and momenutm simultaneously with a step size η. This is **eulers method**

$$v_{t+1} = v_t + \eta \frac{dv}{dt}(\theta_t, v_t) = v(t) - \eta \frac{\partial \mathcal{E}(\theta_t)}{\partial \theta} \tag{352}$$

$$\theta_{t+1} = \theta_t + \eta \frac{d\theta}{dt}(\theta_t, v_t) = \theta_t + \eta \frac{\partial \mathcal{K}(v_t)}{\partial v} \tag{353}$$

The **modified Euler Method** is slightly more accurate and involves first updating the momentum, and then updating the position using the new momentum.

The **leapfrog integrator** is a symmetrized version of modified euler. We first perform "half" a momentum update, then a full update of the position, then finaly "half" update on the momentum:

$$v_{t+0.5} = v_t - \frac{\eta}{2} \frac{\partial \mathcal{E}(\theta_t)}{\partial \theta} \tag{354}$$

$$\theta_{t+1} = \theta_t + \eta \frac{\partial \mathcal{K}(v_{t+0.5})}{\partial v} \tag{355}$$

$$v_{t+1} = v_{t+0.5} - \frac{\eta}{2} \frac{\partial \mathcal{E}(\theta_{t+1})}{\partial \theta} \tag{356}$$

The Hamiltonian Monte Carlo Algorithm

The target distribution has the form:

$$p(\theta, v) = \frac{1}{Z} exp[-\mathcal{H}(\theta, v)] = \frac{1}{Z} exp[-\mathcal{E}(\theta) - \frac{1}{2} v^T \Sigma v] \tag{357}$$

The marginal distribution over the latent variables of interest is:

$$p(\theta) = \int p(\theta, v) dv = \frac{1}{Z_q} e^{-\mathcal{E}(\theta)} \int \frac{1}{Z_p} e^{-\frac{1}{2} v^T \Sigma v} dv = \frac{1}{Z_q} e^{-\mathcal{E}(\theta)} \tag{358}$$

Suppose the previous state of the Markov chain is (θ_{t-1}, v_{t-1}). To sample the next state, we set the initial position to $\theta_0' = \theta_{t-1}$ and sample a random momentum$v_0' \sim \mathcal{N}(0, \Sigma)$. We then initialize a random trajectory in phase space starting at (θ_0', v_0') and follow for L leapfrog steps, until we get to the final proposed state $(\theta^*, v^*) = (\theta_L^*, v_L^*)$.

If we simulated correctly, the energy should be same at the end, else we reject the sample.

The HMC acceptance probability is:

$$\alpha = min(1, \frac{p(\theta^*, v^*)}{p(\theta_{t-1}, v_{t-1})} = min(1, exp[-\mathcal{H}(\theta^*, v^*) + \mathcal{H}(\theta_{t-1}, v_{t-1})] \tag{359}$$

Tuning HMC

We need to specify three hyperparameters for HMC: number of leapfrog steps L, step size η, and covariance Σ.

Riemann Manifold HMC

If we let the covariance matrix change as we move position, so Σ is a function of θ the method is known as **Riemann Manifold HMC** since the moves follow a curved manifold rather than the flat manifold induced by a constant Σ.

A natural choice for the covariance is the Hessian at the current location to capture local geometry:

$$\Sigma(\theta) = \nabla^2 \mathcal{E}(\theta) \tag{360}$$

This might not always be positive definite, so we can use the Fisher information matrix:

$$\Sigma(x) = -\mathbb{E}_{p(x|\theta)}[\nabla^2 logp(x|\theta)] \tag{361}$$

MCMC Convergence

One of the simpelst approaches to assessing if the method has converged is to run multiple chains from very different overdispered starting points, and to plot the samples of some quantity of interest. This is called a **trace plot**. If the chain has mixed, it should have forgotten where it started from, so the trace plots should converge to the same distribution and thus overlap.

Estimated Potential Scale Reduction

If one or more chains has not mixed well, then the variance of all the chains combined together will be higher than the variance of the initial chains.

We can therefore compute the between sequence and within sequence variance and use the **r-hat** ratio to ensure convergence.

Effective Sample size

Even if the chain mixed, the samples are correlated and hence we need to draw a lot of them to get a reliable estimate.

Prediction

Basics

We want o predict ouputs y from inputs x using some function f that is estimated from labeled training set $\mathcal{D} = \{(x_n, y_n) : n = 1, .., N\}$ for $x_n \in \mathcal{X} \subset \mathbb{R}^D$, $y_n \in \mathcal{Y} \subset \mathbb{R}^C$.

We can model our uncertainty about the correct output for a given input using a conditional probability model $p(y|f(x))$.

Parametric models have a fixed number of parameters independent of the size of the training set. They are usually less flexible, but are faster to use for prediction.

Non-parametric models have a variable number of parameters that grows with the size of the training set. They are usually more flexible, but are slower to use for prediction.

Most non-parametric models are based on comparing a test input x to some or all of the stored training examples $\{x_n, n = 1, ..., N\}$, using some form of similarity $s_n = \mathcal{K}(x, x_n) \geq 0$ and then predicting the ouput using some weighted combination of training labels:

$$\hat{y} = \sum_{n=1}^{N} s_n y_n \tag{362}$$

For example, this is how gaussian processes or KNN works.

Most parametric models have the form $p(y|x) = p(y|f(x; \theta))$ where f is some kind of function that predicts the parameters of the ouput distributions (eg. mean or logits).

If f is a linear function of θ for example $f(x; \theta) = \theta^T \phi(x)$ for some *fixed* feature transform ϕ, then the model is called a **generalized linear model**.

If f is non-linear but differentiable function of θ then it is common to represent f using a neural network.

Other predictive models such as decision trees and random forests are discussed in prequal. TODO

Model Fitting Using ERM, MLE and MAP

We discuss some methods for fitting parametric models. The most common way is to use **maximum likelihood estimation** which amounts to solving the following optimization problem:

$$\hat{\theta} = argmax_{\theta \in \Theta} p(\mathcal{D}|\theta) = argmax_{\theta \in \Theta} logp(\mathcal{D}|\theta) \tag{363}$$

If the dataset is N iid data samples, the likelihood decomposes into a product of terms:

$$p(\mathcal{D}|\theta) = \prod_{n=1}^{N} p(y_n|x_n, \theta) \tag{364}$$

and thus we can minimize the following scaled **negative log likelihood**

$$\hat{\theta} = argmin_{\theta \in \Theta} \frac{1}{N} \sum_{n=1}^{N} [0logp(y_n|x_n, \theta)] \tag{365}$$

We can generalize this by replacing the **log loss** $l_n(\theta) = -logp(y_n|x_n, \theta)$ with a more general loss function to get $\hat{\theta} = argmin_{\theta \in \Theta} r(\theta)$ where $r(\theta)$ is the **empirical risk**:

$$r(\theta) = \frac{1}{N} \sum_{n=1}^{N} l_n(\theta) \tag{366}$$

Which is called **empirical risk minimization.**

Empirical risk minimization can easily result in **overfitting** so it is common to add a penalty or regularier term to get:

$$\hat{\theta} = argmin_{\theta \in \Theta} r(\theta) + \lambda C(\theta) \tag{367}$$

where $\lambda \geq 0$ controls the degree of regularization and $C(\theta)$ is some complexity measure.

If we use log loss, and define $C(\theta) = -log\pi_0(\theta)$ where $\pi_0(\theta)$ is some prior distribution, and we use $\lambda = 1$ we recover the **map estimate**:

$$\hat{\theta} = argmax_{\theta \in \Theta} logp(\mathcal{D}|\theta) + log\pi_0(\theta) \tag{368}$$

Model fitting use Bayes, Vi and generalized Bayes

Another way to prevent overfitting is to estimate a *probability distribution over paramters* $q(\theta)$ instead of a point estimate.

That is, we can try to estimate the empirical risk in expectation:

$$\hat{q} = argmin_{q \in \mathcal{P}(\Theta)} \mathbb{E}_{q(\theta)}[r(\theta)] \tag{369}$$

If $\mathcal{P}(\Theta)$ is the space of all probability distributions over paramters, then this solution will converge to a delta function that puts all its probability on the MLE.

We can regularize the problem by preventing the distirbution form moving too far from the prior:

$$\hat{q} = argmin_{q \in \mathcal{P}(\Theta)} \mathbb{E}_{q(\theta)}[r(\theta)] + \frac{1}{\lambda} D_{KL}(q|\pi_0) \tag{370}$$

The solution to this problem is known as **Gibbs Posterior** and is given by:

$$\hat{q}(\theta) = \frac{e^{-\lambda r(\theta)} \pi_0(\theta)}{\int e^{-\lambda r(\theta')} \pi_0(\theta') d\theta'} \tag{371}$$

Suppose we now use the log loss and set $\lambda = N$ to get:

$$\hat{q}(\theta) = \frac{p(\mathcal{D}|\theta)\pi_0(\theta)}{\int p(\mathcal{D}|\theta')\pi_0(\theta') d\theta'} \tag{372}$$

which is the **Bayes posterior**.

This is often intractable, so we can simplify by restricting to a family of distributions $\mathcal{Q}(\Theta) \subset \mathcal{P}(\Theta)$ which gives rise to the objective:

$$\hat{q} = argmin_{q \in \mathcal{Q}(\Theta)} \mathbb{E}_{q(\theta)}[-logp(\mathcal{D}|\theta)] + D_{KL}(q||\pi_0) \tag{373}$$

whic is the formulation of **variational inference**.

Evaluating Predictive Models

It is common to measure performance of a predictive model by using a **proper scoring rule**

Let $S(p_\theta, (y, x))$ be the score for predcitive distribution $p_\theta(y|x)$ when given an event $y|x \sim p^*(y|x)$ where p^* is the true conditional distribution.

If we want to evaluate a beysian model where we marginalize out θ rather than condition on it, we can replace $p_\theta(y|x)$ with $p(y|x) = \int p_\theta(y|x)p(\theta|\mathcal{D})d\theta$.

The expected score is defined by:

$$S(p_\theta, p^*) = \int p^*(x)p^*(y|x)S(p_\theta, (y, x))dydx \tag{374}$$

A **proper scoring rule** satisfies:

$$S(p_\theta, p^*) \leq S(p^*, p^*) \tag{375}$$

with equality $iff p_\theta(y|x) = p^*(y|x)$

This maximizing such a proper scoring rule will force the model to match the true probabilities.

Ex: the log likelihood $S(p_\theta, (y, x)) = logp_\theta(y|x)$ is a proper scoring rule by Gibbs inequality:

$$S(p_\theta, p^*) = \mathbb{E}_{p^*(x)p^*(y|x)}[logp_\theta(y|x)] \leq \mathbb{E}_{p^*(x)p^*(y|x)}[logp^*(y|x)] \tag{376}$$

Therefor minimizing the negative log likelihood (log loss) should result in well calibrate probabilities. However in practice log loss can over-emphasizes tail probabilities.

A common alternative is to use the **Brier Score** which is:

$$S(p_\theta, p^*) = \frac{1}{C}\sum_{c=1}^{C}(p_\theta(y = c|x) - \mathbb{I}(y = c))^2 \tag{377}$$

which is just the squared error of the predictive distribution $p(1 : C|x)$ compared to the one hot label distribution y. Since it is based on squared error, the Brier score is less sensitive to extremely rare or extremely common classes.

Calibration

A model whose predicted probabilities match the empirical frequencies is said to be **calibrated**.

To asses calibration we divide the predicted probabilities into a finite set of bins or buckets and then assess the discrepancy between the empirical probability and predicted probability. These are called **reliability diagrams**, and hte gab between the accuracy and confidence is the **expected calibration error**

Improving calibration of probabilistic classifiers

Let z be the logit and $p = \sigma(z)$ produced by a probabilisitic binary classifier. We wish to convert this to a more calibrated value q. The simplest way is **platt scaling**. The idea is to compute $q = \sigma(az + b)$ where a and b are estimate via maximum likelihood on a validation set.

In multiclass setting we can extend Plat scaling by using matrix scaling.

Platt scaling makes a strong assumption about the shape of the calibration curve. A more flexible non-parametric method is to partition the predicted probabilities into bins p_m and to estimate an empirical probability q_m for each such bin, then replace $p_m \to q_m$. This is called **histogram binning.**

Temperature scaling does:

$$q = softmax(z/T) \tag{378}$$

where $T > 0$ is the temperature parameter which can be estimated by maximum likelihood on a validation set. The effect of this temperature parameter is to make the distribution less peaky. Note it does not effect which class is most probable.

Beyond evaluating marginal probabilities

Assessing properties of the marginal predictive distribution $p(y|x)$ may be insufficient to distinguish between a good and bad model, especially in the context of online learning and sequential decision making.

It is important to evaluate joint predictive distributions when assessing predictive models, for example to evaluate the posterior predictive distributions over τ outcomes.

Conformal Prediction

Conformal prediction is a way to create prediction intervals or sets with guarenteed frequentist coverage probability from any predictive method $p(y|x)$. This can be seen as a form of **distribution free uncertainty**

quantification since it works without making assumptions (beyond exchangeability of the data) about the true data generating process or the form of the model.

We start with a heuristic notion of uncertainty such as softmax score for a classification problem or variance for regresion problem and use it define a **conformal score** $s(x, y) \in \mathbb{R}$ which measures how badly output y *conforms to* x where larger values of the score are less likely.

Next we apply this score to a **calibration set** of an labeled examples that was not used to train f, to get $\mathcal{S} = \{s_i = s(x_i, y_i) : i = 1, .., n\}$ The user specifies a desired confidence threshold α, say $= 0.1$, and we then compute the $(1 - \alpha)$ quantile $\hat{q}$ of $\mathcal{S}$.

Finally, given a new test input x_{n+1} we compute the prediction set to be:

$$\mathcal{T}(x_{n+1}) = \{y : s(x_{n+1}, y) \leq \hat{q}\} \tag{379}$$

Intuitively we include all the outputs y that are plausible given the input.

Generalized Linear Models

A **generalized linear model** is a conditional version of an exponential family distribution:

$$p(y_n|x_n, w, \sigma^2) = exp[\frac{y_n\eta_n - A(\eta_n)}{\sigma^2} + logh(y_n, \sigma^2)] \tag{380}$$

where $\eta_n = w^T x_n$ is the naturla parameter for the distribution, $A(\eta_n)$ is the log normalizer, $\mathcal{T}(y) = y$ is the sufficient statistic and σ^2 is the disperion term.

The mean and variance of the response variable are:

$$\mu_n = \mathbb{E}[y_n|x_n, w, \sigma^2] = A'(\eta_n) = l^{-1}(\eta_n) \tag{381}$$

$$\mathbb{V}[y_n|x_n, w, \sigma^2] = A''(\eta_n)\sigma^2 \tag{382}$$

we denote the mapping from the linear inputs to the mean of the output using $\mu_n = l^{-1}(\eta_n)$ where the function l is known as the **link function** and its inevrse l^{-1} known as the **mean function**.

We usually write:

$$l(\mu_n) = \eta_n = w^T x_n \tag{383}$$

Linear Regression

Linear regression has the form:

$$p(y_n|x_n, w, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{1}{2\sigma^2}(y_n - w^T x_n)^2) \tag{384}$$

and hence:

$$logp(y_n|x_n, w, \sigma^2) = -\frac{1}{2\sigma^2}(y_n - \eta_n)^2 - \frac{1}{2}log(2\pi\sigma^2) \tag{385}$$

where $\eta_n = w^T x_n$ and thus $\mathbb{E}[y_n] = \eta_n = w^T x_n$ and $\mathbb{V}[y_n] = \sigma^2$

Binomial regression

If the response variable is the number of successes in N_n trails $y_n \in \{0, .., N_n\}$ we can use **binomial regression** defined as:

$$p(y_n|x_n, N_n, w) = Bin(y_n|\sigma(w^T x_n), N_n) \tag{386}$$

and we see that **binary logistic regression** is the special case when $N_n = 1$

The log pdf is:

$$logp(y_n|x_n, N_n, w) = y_n log\mu_n + (N_n - y_n)log(1 - \mu_n) + Const \tag{387}$$

and hence $\mathbb{E}[y_n] = N_n\mu_n$ and $\mathbb{V}[y_n] = N_n\mu_n(1 - \mu_n)$

Maximum Likelihood Estimation

Generalized linear models can be fit using similar methods to those that we used to fit logistic regression, in particluar hte negative log-likelihood has the form:

$$NLL(w) = -logp(\mathcal{D}|w) = -\frac{1}{\sigma^2}\sum_{n=1}^{N} l_n \tag{388}$$

where $l_n = \eta_n y_n - A(\eta_n)$, and $\eta_n = w^T x_n$.

Note: the hessian is positive definite, hence the negative log liklihood is convex, so the MLE for a generalized linear model is unique.

For small datasets we can use the **itratively reweighted least squares** which is a form of Newtons method ot compute the MLE. For large datasets, we can use SGD.

Bayesian Linear Regression

Noise Variance is known

The conjugate prior for linear regression has the form :

$$p(w) = \mathcal{N}(w|w', \Sigma') \tag{389}$$

and we often take the prior mean to be zero, and prior covariance to be isotopic $\tau^2 I$

In this case, the posterior mean becomes:

$$\hat{w} = (\frac{\sigma^2}{\tau^2} I + X^T X)^{-1} X^T y \tag{390}$$

and if we define $\lambda = \frac{\sigma^2}{\tau^2}$ we get **ridge regression** which optimizes:

$$\mathcal{L}(w) = \frac{1}{2}||Xw - y||_2^2 + \lambda||w||^2 = \frac{1}{2}\sum_{n=1}^{N}(y_n - w^T x_n)^2 + \lambda||w||^2 \tag{391}$$

Noise Variance is unknown

If w and σ^2 are both unknown, the conjugate prior for w has the form: $\text{p(w—}\sigma^2) = \mathcal{N}(w|w', \sigma^2\Sigma'$

For the noise variance σ^2 the conjugate prior is based on the *inverse gamma distribution*

and hence the joint conjugate prior is the **normal inverse gamma distribution**.

The resultant posterior:

$$p(w, \sigma^2|\mathcal{D}) = NIG(w, \sigma^2|w'', \Sigma'', a'', b'') \tag{392}$$

$$w'' = \Sigma''(\Sigma'^{-1}W' + X^T y) \tag{393}$$

$$\Sigma'' = (\Sigma'^{-1} + X^T X)^{-1} \tag{394}$$

$$a'' = a' + \frac{N}{2} \tag{395}$$

$$b'' = b' + \frac{1}{2}(w'^T \Sigma'^{-1} w' + y^T y - w'^T \Sigma'^{-1} w') \tag{396}$$

Posterior predictive distribution We usually care more about uncertainty and accuracy of our predictions, not our parameter estimates.

Given N' new test inputs $\overline{X}$ we have the posterior predictive distribution:

$$p(\overline{y}|\overline{X}, \mathcal{D}) = \int\int p(\overline{y}|\overline{X}, w, \sigma^2) p(w, \sigma^2|\mathcal{D} dw d\sigma^2 = ... = \mathcal{T}(\overline{y}|\overline{X}w'', \frac{b''}{a''}(I_{N'} + \overline{X}\Sigma''\overline{X}^T, 2a'') \tag{397}$$

the expectation is obvious, the first term is due to measurement noise, the second ter is due to uncertainty in our weights, which varies based on how close the test inputs are to the training data. The predictive distribution is therefore wider than using gaussian prior with fixed σ^2 because here we are taking into account uncertainty about σ^2.

Logistic Regression

Logistic regression is a widely used discriminative classificatio model that maps input vectors $x \in \mathbb{R}^D$ to a distribution over class labels $y \in \{1, ..., C\}$. If $C = 2$ then we have **binary logistic regression**, otherwise we have **multinomial logistic regression**.

Binary Logistic regression

$y \in \{0, 1\}$ and

$$p(y|x; \theta) = Ber(y|\sigma(w^T x + b)) \tag{398}$$

where w are the weights, b is the bias and σ is the **sigmoid or logistic function**

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{399}$$

Let $\eta_n = w^T x_n + b$ be the **logits** for example n, and $\mu_n = \sigma(\eta_n) = p(y = 1|x_n)$ be the mean of the output. Then we can write the log likelihood as the negative cross entropy:

$$logp(\mathcal{D}|\theta) = log \prod_{n=1}^{N} \mu_n^{y_n} (1 - \mu_n)^{1-y_n} = \sum_{n=1}^{N} y_n log\mu_n + (1 - y_n) log(1 - \mu_n) = \sum_{n=1}^{N} y_n \eta_n - \sum_{n=1}^{N} log(1 + e^{\eta_n}) \tag{400}$$

The **logsumexp** rule means that $log(1 + e^a$ is often implemented:

$$np.log1p(np.exp(a)) \tag{401}$$

Multinomial logistic regression

Multinomial logistic regression is a discriminative classification model like:

$$p(y|x, \theta) = Cat(y|softmax(Wx + b)) \tag{402}$$

where $x \in \mathbb{R}^D$ is the input vector $y \in \{1, .., C\}$ is the class label, $W \in C \times D$ is the weight matrix and b is a C-dimensional bias vector. The **softamx function** is defined as:

$$softmax(a) = [\frac{e^{a_1}}{\sum_{c'=1}^{C} e^{a_{c'}}}, ..., \frac{e^{a_C}}{\sum_{c'=1}^{C} e^{a_{c'}}}] \tag{403}$$

If we define the logits as $\eta_n = Wx_n + b$ the probabilities as $\mu_n = softmax(\eta_n)$ and let y_n be the one hot encoding of the label y_n then the log likelihood can be written as the negative cross entropy:

$$logp(\mathcal{D}|\theta) = log \prod_{n=1}^{N} \prod_{c=1}^{C} \mu_{nc}^{y_n c} = \sum_{n=1}^{N} \sum_{c=1}^{C} y_{nc} log\mu_{nc} \tag{404}$$

It is standard to use Gaussian priors for the weights in logistic regression model, with prior mean 0.

We should standardize inputs, and scale the variance of the prior by $\frac{1}{\sqrt{D}}$ because otherwise the induced prior is initially flat, but eventually becomes skewed towards the extreme values 0, 1.

There is no tractable prior that is conjugate to the logistic likelihood, even if we use Gaussian prior cannot compute posterior analytically, unlike with linear regression.

Like in MLE, where we have closed form solution for linear regression but not for logistic regression.

Deep Neural Networks

A **deep neural network** is any differentiable function that can be expressed as a **computation graph** where the nodes are primitive operations and edges represent numeric data in the form of vectors, matrices or tensors.

We can combine DNN with probabilistic models in two ways. First, we can use them to define nonlinear functions which are used inside conditional distributions, for example, we can output the softmax of logits outputted by a neural network. The other is we can use a DNN to approximate the posterior distribution.

Building Blocks of Differentiable Circuits

Canonical Examples of Neural Networks

Linear Layers

The most basic building block of a DNN is a single **neuron** which corresponds to a real valued signal y computed by multiplying a vector valued input signal x by a weight vector w and then adding bias b:

$$y = f(x;\theta) = w^T x + b \tag{405}$$

where $\theta = (w, b)$ are the paramters for the function f.

It is common to group a set of neurons together into a **layer**, so we can represent a layer with D units as a vector $z \in \mathbb{R}^D$. We can transform an input vector of activations x into an ouput vector y be multiplying by a weight matrix W and adding an offset b:

$$y = f(x;\theta) = Wx + b \tag{406}$$

which is called a **fully connected layer**.

It is common to prepend the bias bector onto the first column of the weight matrix, and to append a 1 to the vector x wo we can write $y = \tilde{W}^T \tilde{x}$ where $\tilde{W} = [W, b]$ and $\tilde{x} = [x1]$

Non-linearities

A stack of linear layers is equivalent ot a single linear layer, so to get nonlinear expression we can transform each layer by passing it elementwise throuhg a nonlinear funcition called an **activation function**:

$$y = \phi(x) = [\phi(x_1), ..., \phi(x_D)] \tag{407}$$

- Sigmoid: $\sigma(a) = \frac{1}{1+e^{-a}} \in [0, 1]$
- Hyperbolic Tangent: $tanh(a) = 2\sigma(a) = 2\sigma(2a) - 1 \in [-1, 1]$
- Softplus $\sigma_+(a) = log(1 + e^a) \in [0, \infty]$
- Relu: $max(a, 0) \in [0, \infty]$
- Leaky Relu: $max(a, 0) + \alpha min(a, 0) \in [-\infty, \infty]$
- ExpRelu: $max(a, 0) + min(\alpha(e^a - 1, 0)) \in [-\infty, \infty]$
- Swish: $a\sigma(a) \in [-\infty, \infty]$

Convolution Layers

When dealing with image data, we can apply the same weight matrix to each local patch of the image, in order to reduce the number of parameters. If we slide this weight matrix over the image and add up the result, we get a **convolution**.

The weight matrix is called a **kernel**.

Let $X \in \mathbb{R}^{H \times W}$ be an input image, and $W \in \mathbb{R}^{h \times w}$ be the kernel of weights.

The output denoted by $Z = XconvW$ is:

$$Z_{i,j} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} x_{i+u,j+v} w_{u,v} \tag{408}$$

Essentially, we compare a local patch of x of size $h \times w$ centered at (i, j) to the filter w. Often the output size can be smaller than the input size which we can resolve by using **padding**.

If we repeat this process for multiple layers of inputs, and by using multiple filters, we can generate multiple layers of output.

If we have C input channels, and we want to map it to D output features channels, then we define D kernels each of size $h \times w \times C$ where h, w is the size of each kernel. The dth output feature map is obtained by convoling all C input feature maps with the dth kernel, and then adding the results elementwise:

$$z_{i,j,d} = \sum_{u=0}^{h-1} \sum_{v=0}^{w-1} \sum_{c=0}^{C-1} x_{i+u,j+v,c} w_{u,v,c,d} \tag{409}$$

which is called a **convolution layer.**

Convollution layers share weights across location, which leads to **shift equivariance**. In some cases, we want the output to be the same, no matter where the input pattern occurs this is called **shift invaraince** and can be obtained by using a **pooling layer** which computes the maximum or average value in each local aptch of teh input. Note that the pooling layers have no learnable parameters.

Residual connections If we stack large number of nonlinear layers together, the signal may get squared to zero or explode to infinity, depending on the magnittud eof the weights and nature of nonlinearities. This can plague gradients that are passed backwards through the network.

To reduce this effect, we can add **skip connections** or **residual connections** which allow the signal to skip one or more lahyers, hence preventing it from being modified.

Normalization Layers

To learn an input-output mapping, it is often best if the inputs are standardized meaning they have zero meand and unit standard deviation. This ensures that the required magnitude of the weights is amll and comparable across dimensions.

The most common normalization is **batch normalization** which relies having acces to abatch of $B > 1$ input examples.

Dropout

Neurons nets have million of paramaters that can sometimes overfit, especially on small datasets. We can apply regularizers, but an effective approximation to bayesian approach is **dropout** in which edges are rnaomdly ommited each time the network is used.

If w_{lij} is the weight of the edge from node i in layer l-1 to node j in layer l then we can replace it with $\theta_{lij} = w_{lij}\varepsilon_{li}$ where $\varepsilon_{li} \sim Ber(1-p)$ where p is the drop probability, and 1-p is the keep probability. This if we sample $\varepsilon_{li} = 0$ then all of the weights going out of unit i in layer $l-1$ into any j in layer l will be set to 0.

Due to the staochastisticty, we end up with an **ensemble of networks** during training, each with slightly different sparse graph structures.

At test time, we usually turn the dropout noise off, so the model acts deterministically.

To ensure the weights ahve the same expectation at test time as they did during training, at test time we should use $\mathbb{E}[\theta_{lij}] = w_{lij}\mathbb{E}[\varepsilon_{li}]$. For Bernoulli noise we have $\mathbb{E}[\varepsilon] = 1 - p$ so we should multiple the weights by the keep probability $1 - p$ before making prediction.

If we use dropout at test time it is called **Monte Carlo dropout**.

Attention Layers

We have a learned embedding W along with the stored examples X to create a stored set of **keys** $K = W^K X \in \mathbb{E}^{n \times d_k}$. Similary we have a learned embedding along with the stored output matrix Y to get a set of stored **values** $V = W^V Y \in \mathbb{R}^{n \times d_v}$. Finally, we embed the input to create a **query** $q = W^Q x \in \mathbb{R}^{d_k}$.

To ensure the output is differentiable in the inputs, we replace the fixed kernel function with a **soft attention layer**:

$$Attn(q, (k_1, v_1), ..., (k_n, v_n)) = Attn(q, (k_{1:n}, v_{1:n}) = \sum_{i=1}^{n} \alpha + i(q, k_{1:n}) v_i \tag{410}$$

where $\alpha_i(q, k_{1:n})$ is the **ith attention weight** that satisfy $0 \leq \alpha_i(q, k_{1:n}) \leq 1$ for each i and $\sum_i \alpha_i(q, k_{1:n}) = 1$.

The attention weights can be computed from an **attention score function** $a(q, k_i) \in \mathbb{R}$ that computes the similarity of query q to key k_i. For example, we can use **scaled dot product attention** which has the form:

$$a(q, k) = \frac{q^T k}{\sqrt{d_k}} \tag{411}$$

where the scaling helps reduce the dependence of the output on the dimensionality of the vectors.

Given the scores, we can compute the attention weights using the softmax:

$$\alpha_i(q, k_{1:n}) = softmax_i([a(q, k_1), ..., a(q, k_n)]) = \frac{exp(a(q, k_i))}{\sum_{j=1}^{n} exp(a(q, k_j))} \tag{412}$$

Sometimes, we want to restrict attention to a subset of the dictionary, corresponding to valid entries. Ex: we want to pad sequences to a fixed length for efficient minibatching, in which case we should maks out the padded location. This is called **masked attention**.

We can implement this efficiently by setting the attention score for the masked entries to a large negative number, so that the corresponding softmax weight will be 0.

In practice, we usually deal with minibatches of n vectors at a time. Let the correspdongin matrices of queries, keys and values by denoted by $Q \in \mathbb{R}^{n \times d_k}, K \in \mathbb{R}^{n \times d_k}, V \in \mathbb{R}^{n \times d_v}$ and let:

$$z_j = \sum_{i=1}^{n} \alpha_i(q_j, K) v_i \tag{413}$$

be the jth output corrresponding to the jth query.

We can compute all outputs $Z \in \mathbb{R}^{n \times d_v}$ in parallel using:

$$Z = Attn(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{414}$$

where the softmax function is applied row-wise

To increase the flexibility of the model, we often use **multi-head attention**. Let ith head be:

$$h_i = Attn(QW_i^Q, KW + i^K, VW_i^V \tag{415}$$

where $W_i^Q \in \mathbb{R}^{d \times d_k}, W_i^K \in \mathbb{R}^{d \times d_k}, W_i^V \in \mathbb{R}^{d \times d_v}$ are linear projection matrices. We define the output of the multihead attention layer to be:

$$Z = MultiHeadAttention(Q, K, V) = Concat(h_1, .., .h_h)W^O \tag{416}$$

where h is the number of heads, and $W^O \in \mathbb{R}^{hd_v \times d}$.

Having multiuple heads can increase performance of the layer in the event that some weight matrices are poorly initialized. Often, after training, we can remove all but one of the heads.

When the output of one attention layer is used as input to another, the method is called **self-attention** which is the basis of the **transformer model**.

Recurrent Layers

We can make the model **stateful** by augmenting the input x with the current state s_t and then computing the output and the new state using some kind of function:

$$(y, s_{t+1}) = f(x, s_t) \tag{417}$$

this is called a **recurrent layer** which is the bases of **recurrent neural networks**.

In a vanilla RNN, teh function f is a simple MLP, but it may also use attention.

Implicit Layers

Implcit layers specify the output indirectly in terms of a constraint:

$$y \in argmin_y f(x, y) s.t. g(x, y) = 0 \tag{418}$$

We may need to run an inner optimization or ODE solver,. We gain the fact that the inner computations do not need to be stored explicitly, saving memory. Furthermore, once the solution is found, we can propogate gradients through the whole layer by leveragin the **implicit function theorem**. This lets us use higher level primitives inside an end-to-end framework.

Implicit Functons and Automatic Differentiation

How do we perform efficient automatic differentiation of functions defined implicitly by fixed point equations?

We say $z \in \mathbb{R}^n$ is a **fixed point** of $f : \mathbb{R}^n \to \mathbb{R}^n$ if :

$$z = f(z) \tag{419}$$

More generally, we might have a *paramaterized function* $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^p$ taking parameter vector $a \in \mathbb{R}^p$, and then we'd write a fixed point as:

$$z = f(a, z) \tag{420}$$

which corresponds to have a paramterized system of equations.

The simplest way to compute numerical fixed points is **naive forward iteration** where we iterate $z_{k+1} = f(z_k)$ until we are sufficiently close.

This only would work in particular forms (contractions) of f, and good initial starting points.

We can use **newton iteration** where we use derivative information about f to take a smarter step (at the cost of using more computation per step).

We can differentiate through these solvers, but this would be extremely inefficient since an autograd framework will need to store iterative values during each iteration, leading to high memory consumption. We can use the **implicit function theorem to help us**.

Theorem 0.101. ***The implicit function theorem on*** $\mathbb{R}^n$

Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ *and* $a_0 \in \mathbb{R}^p, z_o \in \mathbb{R}^n$ *be such that:*

- $f(a_0, z_0) = 0$
- f *is* C^1 *with non singular Jacobian* $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$

then there exists open sets $S_{a_0} \subset \mathbb{R}^p$ *and* $S_{z_0} \subset \mathbb{R}^n$ *containing* a_0, z_0 *and a unique continuous function* $z^* : S_{a_0 \to S_{z_0}}$ *such that:*

- $z_0 = z^*(a_0)$
- $f(a, z^*(a)) = 0 \forall a \in S_{a_0}$
- z^* *is* C^1 *on* S_{a_0}

We can think of a *solution mapping function* z^* which satisfies:

$$f(a, z^*(a)) = 0 \forall a \in S_{a_0} \tag{421}$$

and since both sides are function of a, we can differentiate and evaluate at (a_0, z_0):

$$\partial_0 f(a_0, z_0) + \partial_1 f(a_0, z_0) \partial z^*(a_0) = 0 \tag{422}$$

so that:

$$\partial z^*(a) = -[\partial_1 f(a_0, z_0)]^{-1} \partial_0 f(a_0, z_0) \tag{423}$$

so that the Jacobian of the solution mapping can be expressed just int erms of Jacobian of f at the solution poiont (a_0, z_0).

Thus, no matter how we solve the equation to compute z_0, we can still compute the Jacobian using just derivative information at the solution point, and help us avoid differentiating through our iterative solvers!

Recall that Automatic differentiation is built on two transformations: **Jacobian vector products (JVP)** and **Vector-Jacobian produts** (VJP).

The **Jacobian vector products** model the mapping:

$$(x, v) \mapsto (f(x), \partial f(x) v) \tag{424}$$

This answers questions like:

- at a given input point x, if we nudge the input by a vector v, how does the ouput change to first order?
- What are the first two terms of the Taylor series: $f(x + v) = f(x) + \partial f(x) v + \mathcal{O}(||v||^2)$

In programs, JVP underlie **forward mode auto-diff**, in the sense that if your autodiff system says it implements forward mode, then it gives you ways to compute JVP for a function in your programming language.

In contrast, **vector Jacobian products** lets us evaluate Jacobian matrices one *row* at a time, and model the mapping:

$$(x, w) \mapsto (f(x), w^T \partial f(x)) \tag{425}$$

which answers questions like:

- at a given input x if we have a vector w that represents a scalar valued linear function on perturbations Δy of the output (eg. representing how a scalar-valued loss function changes for small changes to the ouput), then what is a vector reprsenting the correspdong liner function on perturbation Δx to the input? I.e., what is $\lambda \in \mathbb{R}^n$ such that:

  $$\langle w, \Delta y \rangle = \langle w, \partial f(x) \Delta x \rangle = \langle \lambda, \Delta x \rangle \tag{426}$$

  for any Δx where we've defined $\Delta y = \partial f(x) \Delta x$

In programs, VJP underlie **reverse mode autodiff**. The reason reverse mode is so omnipresent in ML is due to the VJP relationship to the gradient of a scalar valued function, and the importance of gradient based optimization of scalar-valued loss function on the parameters of a neural network. Indeed, if $l : \mathbb{R}^n \to \mathbb{R}$ and we have n parameters in a network, the Jacobian matrix of l has one row and like a billion columns, hence we would rather compute one row at a time VJP.

Neural ODEs

An ODE initial value problem has the form:

$$y'(t) = f(y(t), t, \theta), y(0) = y_0 \tag{427}$$

Common Examples of Neural Networks

A **feedforward neural network** consists of a series of L linear layers, combined with elementwise nonlinearities:

$$f(x, \theta) = W_{L\phi L}(W_{L-1\phi L-1}(\cdots\phi_1(W_1 x_\cdot \cdots)) \tag{428}$$

For classification, the final nonlinearity is usually the softmax, or we can keep them as logits and have the loss functions vonvert to log probabilities internally.

A **convolutional neural network** consists of a series of convolution layers, pooling layers, linear layers and nonlinearities. Architectures like resnet add skip connections, normalization layers

An **autoencoder** is a neural netwokr that maps inputs x to a low-dimensional latent space using an **encoder** $z = f_e(x)$ and then attempts to reconstruct the inputs using a **decoder** $\hat{x} = f_d(z)$. The model is trained to minimize:

$$\mathcal{L}(\theta) = ||f_d(f_e(x)) - x||_2^2 \tag{429}$$

For image data, we can let the encoder be a convolution network, and the decoder be a transpose convolutional network.

A **recurrent neural network** is a network with a recurrent layer, which defines the followign probability distribution over sequences:

$$p(y_{1:T} = \sum_{h_{1:T}} p(y_{1:T}, h_{1:T}) \tag{430}$$

where h_t s a deterministic hidden state, computed from the last hidden state and last ouput using $f(h_{t-1}, y_{t-1})$. At training time y_{t-1} is observed but at prediction time, generation.

In a vanilla RNN, the function f is an MLP, but we can also use attention to selectively update parts of the state vector based on similarity between the input the previous state as in GRU or $LSTM$ cells. We can also make the model conditional sequence model by feeding in extra inputs to f.

RNN process one token at a time, and hence are autoregressive. To do well in sequence-sequence modelling, we need to learn a contextual embedding of each word. In RNN, the embedding of the word at location t, z_t depends on the hidden state of the network s_t which may be a lossy summary of all the previously seen words. An alternative approach is to compute z_t as a direct function of all the other words in the sentence, by using the attention operator rather than a hidden state. This is called an **encoder only transformer** and is used by models such as BERT. It is also possible to create a **decoder only transformer** in which each output y_t only attends to all the previously generated outputs $y_{1:t-1}$, implemented by using masked attention. This is used by *generative language models* such as GPT.

We can combine the encoder and decoder to create a conditional sequence-sequence model $p(y_{1:T_y}|x_{1:T_x})$. Large transformers are flexible sequence-to-sequence function approximators given enough data.

The *encoder* takes the embedded input tokens X and passes them through a multihead attention layer, and the output Z is added to the input X as a residual connection.

SO if $X = (x_1, ..., x_n)$ we compute:

$$x_i = x + i + \sum_{j=1}^{n} K_{ijW_V x_j} \tag{431}$$

where $K = softmax(A), A_{ij} = (W_Q x_i)^T (W_k x_j)$.

The output of self attention is then passed into a layer normalization layer, which normalizes and learns an affine transformation for each dimensions, to ensure all hidden units have similar magnitide, then the output vectors at each location are mapped through an MLP.

The overall encoder is N copies of this encoder block, and the result is an encoding $H_x \in \mathbb{R}^{T_x \times D}$ of the input, where T_x is the number of input tokens, and D is the dimensionality of the attention vectors.

Once the input is encoded, the output is generated by the *decoder*.

The first part of the decoder is the decoder attention block, that attends to all previously generated tokens $y_{1:t-1}$ and computes the encoding $H_y \in \mathbb{R}^{T_y \times D}$. This block uses masked attention so that outputs at time t can only attend to locations prior to t.

The second part of the decoder is the *encoder-decoder* attention block, that attends to both the encoding of the input H_x and the previously generated outputs H_y. These are combined to form $Z = MultiHeadAttention(Q = H_y, K = H_x, V = H_x)$ which compares the output to the input. The joint encoding of state z is passed through and MLP. The full decoder repeats this decoder block N times.

At the end of the decoder, the final output is mapepd to a sequence of T_y output logits via a final linear layer.

The encoder needs to use **positional encoding** application to the input tokens $x \in \mathbb{R}^{T_x \times D}$.

Graph Neural Networks

Let there by N nodes, each associated with a feature vector to create the matrix $V \in \mathbb{R}^{N \times D_v}$. We also have a set of E edges, each associate with a feature vector forming the matrix $E \in \mathbb{R}^{E \times D_e}$. We also have a global feature vector $u \in \mathbb{R}^u$ representing overall properties of the graph.

The topology of the graph can be represetned by an adjacency matrix, but a more compact representation is the adjancey list.

A basic GNN layer updates the embedding vectors associated with the node, edges, and global graph.

The update functions are typicaly some form of neural network that are applied independently to each embedding vector.

To leverage the graph structure, we cna combine informaiton using a **pooling operation**, where for each node n, we extract the feature vectors associated with its edges, and combine it with its local feature vector using a permutaiton invariant operation such as summation or averaging. We denote the pooling by $\rho_{E_n \to V_n}$. We can similar pool from ndoes to edges $\rho_{V_n \to E_n}$ or from nodes to globals $\rho_{V_n \to U_n}$.

Instead of transforming each vector independently and then pooling, we can first pool the information for each node or edge, and then update its vector representation. For node i, we **gather** informaiton from all the neighboring nodes $\{h_j : j \in nbr(i)\}$ we **aggregate** these vectors with the local vector using an operation such as sum, and then we compute the new state using an **update function** such as:

$$h'_i = Relu(Uh_i + \sum_{j \in nbr(i)} Vh_j) \tag{432}$$

After K message passingl ayers, each node will have received information from neighbors which are K steps away in the graph.

Bayesian Neural Networks

Deep neurla networks are usually trained using a penalized maximum likelihood objective to find a single setting of parameters. However, large flixble models like neural networks can represent many functions, corresponding to different parameter settings, which fit the training data well, yet generalize in different ways. This can be done by computing the posterior predictive distribution using Bayesian model averaging:

$$p(y|x,\mathcal{D}) = \int p(y|x,\theta)p(\theta|\mathcal{D})d\theta \tag{433}$$

where $p(\theta|\mathcal{D}) \propto p(\theta)p(\mathcal{D}|\theta)$.

The main challenge in applying Bayesian inference to DNN are specifiying suitable priors, and efficiently computing the posterior, which is challenging due to the large number of parameters.

Priors for Bayesian Neural Networks

We need to specify a prior $p(\theta)$.

The most common is **gaussian prior** on the weights and biases that is factorized:

$$W_l \sim \mathcal{N}(0, \alpha_l^2 I), b_l \sim \mathcal{N}(0, \beta_l^2 I) \tag{434}$$

THe **Xavier intiialization** is to set:

$$\alpha_l^2 = \frac{2}{n_i n + n_o ut} \tag{435}$$

where $n_i n$ is the fan-in of a node in level l (number of weights coming into a neuron) and $n_o ut$ is the fan-out (number of weights going out of a neuron)

The **LeCun initialization** corresponds to:

$$\alpha_l^2 = \frac{1}{n_i n} \tag{436}$$

We can also use **sparsity promiting priors** such as **Laplace** which encourage most of the weights to be zero.

We can also *learn the prior* using gradient based methods to optimize the marginal likelihood:

$$logp(\mathcal{D}|\alpha,\beta) = \int logp(\mathcal{D}|\theta)p(\theta|\alpha,\beta)d\theta \tag{437}$$

This is known as **empirical bayes** or **evidence maximization**.

Note: the architecture itself is a strong prior on the distribution over functions we model; ex: CNN encode translation equivariance due to its use of convolution

Posteriors for Bayesian Neural Networks

Monte Carlo Dropout

Monte Carlo dropout is a simple method for approximating the Bayesian predictive distribution. The idea of monte carlo dropout is to keep stochastic dropout layers at test time, and keep the random sampling. We can perform S samples, to get S models, an equally weighted average of the predictive distributions for each of these models:

$$p(y|x,\mathcal{D}) \approx \frac{1}{S}\sum_{s=1}^{S} p(y|x,\theta^s) \tag{438}$$

where θ^s is the version of the MAP parameter estimate where we randomly drop out some connections.

An issue is that Monte Carlo dropout does not give proper uncertainty estimates. Although it can be viewed as a form of variational inference, this is only true under a degenerate posterior approximation, corresponding to a mixture of two delta functions, one at 0 (for dropped out nodes) and one at the MLE.

The posterior will not converge to the true posterior (which is a delta function at the MLE) even as the training set size goes to infinity, since we are always dropping out nodes with a constant probability p.

Laplace Approximation

The **laplace approximation** computes a Gaussian approximation to the posterior $p(\theta|\mathcal{D})$ cenetered at the MAP estimate θ^*. The posterior prediction matrix is equal to the Hessian of the negative log joint computed at the mode.

Let $f(x_n, \theta) \in \mathbb{R}^C$ be the prediction function with C outputs, and $\theta \in \mathbb{R}^P$ be the parameter vector.

Let $r(y; f) = \nabla_f logp(y|f)$ be the residual, and $\Lambda(y; f) = -\nabla_f^2 logp(y|f)$ be the per-input noise term.

In addition, let $J \in \mathbb{R}^{C \times P}$ be the Jacobian, $[J_\theta(x)]_{ci} = \frac{\partial f_c(x,\theta)}{\partial \theta_i}$ and $H \in \mathbb{R}^{C \times P \times P}$ be the Hessian, $[H_\theta(x)]_{cij} = \frac{\partial^2 f_c(x,\theta)}{\partial \theta_i \partial \theta_j}$.

Then the gradient and Hessian of the log likelihood are given by:

$$\nabla_\theta logp(y|f(x,\theta)) = J_\theta(x)^T r(y; f) \tag{439}$$

$$\nabla^2_\theta logp(y|f(x,\theta)) = H_\theta(x)^T r(y; f) - J_\theta(x)^T \Lambda(y; f) J_\theta(\theta) \tag{440}$$

Since the network hessian H is usually intractable, it is usually dropped leaving only the Jacobian term.

This is called the **generalized Gauss-Newton** approximation.

The Generalized gauss newton approximation is guarenteed to be positive definite, but this is not true for the original Hessian since the objective is not convex.

Thus, for a gaussian prior $p(\theta) = \mathcal{N}(\theta|m_o, S_o)$ the lapalce approximaiton becomes $p(\theta|\mathcal{D}) = \mathcal{N}(\theta^*, \Sigma_{GCN})$ where:

$$\Sigma^{-1}_{GCN} = \sum_{n=1}^{N} J_{\theta^*}(x_n)^T \Lambda(y_n; f_n) J_{\theta^*}(x_n) + S_0^{-1} \tag{441}$$

Inverting this matrix takes $O(P^3)$ time, so usually an approximation is used.

FOr example, we can use a diagonal approximation, which takes $O(P)$ time.

A more sophisticated approach is called **Kronecker Factored Curvature (KFAC)** which approximates the covariance of each layer using a Kronecker product.

A limitation of Laplace is that the posterior covariance is derived from the Hessian evaluated at the MAP parameters, which makes a *highly local approximation*, therefore only capturing the local characteristics of the posterior at the MAP parameters - and may therefore suffer badly form local optima, providing overly compact or diffuse representation.

Variational Inference

In fixed form variational inference, we choose a distribution for the posterior approximation $q_\psi(\theta)$ and minimize $D_{KL}(q|p)$ with respect to ψ. We often choose a Gaussian approximate posterior, $q_\psi(\theta) = \mathcal{N}(\theta|\mu, \Sigma)$ which lets us use the reparamterization trick to create a low variance estimator of teh gradient of the ELBO.

If we use a Gaussian approximation with a diagonal covariance, then we approximate the distribution of every parameter as an independent univariate gaussian, where the mean is the point estimate, and the variacne captures the ucnertainty. In **Bayes by backprop** we improve by using the reparamterizatino trick to compute lower variance estimate of the ELBO.

Last Layer Methods

A simple approximation is to only be bayesian about the weights in the final layer and to use MAP estimates for all other parameters, which is called **neural linear approximation**.

MCMC Methods

The gold standard method is to use Hamiltonian monte carlo to approximate the posterior, since it does not make strong assumptions about the form of the posterior.

A challenge of applying this is that it requires access to the full training set at each step. Stochastic gradient MCMC methods operate on mini-batches of data, and offer a scalable alternative.

Deep Ensembles

Many conventional approxiamte inference methods focus on approximating the posterior $p(\theta|\mathcal{D})$ in a local neighborhood around one of the posterior modes. In modern deep networks, this is problematic since these deep networks have highly multi-model posteriors, with parameters in different modes giving rise to very different functions.

A simple alternative called **deep ensembles** involves trianing multiple models, and then to approximate the posterior using an equally weighted mixture of delta functions:

$$p(\theta|\mathcal{D}) \approx \frac{1}{M}\sum_{m=1}^{M}\delta(\theta - \hat{\theta}_m) \tag{442}$$

where M is the number of models, and $\hat{\theta}_m$ is the MAP estimate for model m.

In the **multi-swag** method, we fit a gaussian to each local mode, resulting in a *mixture of gaussian*.

Note: deep ensembles are different from standard ensemble methods such as bagging and random forests, which obtain diversity of it spredictors by training them on different subsets of the data (bootstrap resampling). This data perturbation is necessary when the base learner is a convex problem. In deep ensembles, the diversity arises due to *different starting parameters*, different *random seeds*, and *SGD noise*, which induce different solutions due to the nonconvex loss.

We can also remove the M times memory overhead of running M models, by sharing most of the parameters, and only letting each ensemble member m estimate its own local perturbation.

Approximating the posterior predictive distribution

Once we have approximated the parameter posterior $q(\theta) \approx p(\theta|\mathcal{D})$, we can use it to approximate the posterior predictive distribution:

$$p(y|x,\mathcal{D}) = \int q(\theta)p(y|x,\theta)d\theta \tag{443}$$

which we approximate using Monte Carlo:

$$p(y|x,\mathcal{D}) \approx \frac{1}{S}\sum_{s=1}^{S} p(y|f(x,\theta^s)) \tag{444}$$

with $\theta^s \sim q(\theta)$

Tempered and Cold Posteriors

When working with Bayesian neural networks for classification, the likelihood is usually taken to be :

$$p(y|x,\theta) = Cat(y|softmax(f(x;\theta)) \tag{445}$$

where $f(x;\theta) \in \mathbb{R}^C$ returns the logits over the C class labels.

In practice, BNN gives better predictive accuracy if hte likelihood funciton is scaled by some power α, so that we target the **tempered posterior**:

$$p_{tempered}(\theta|\mathcal{D}) \propto p(y|X,\theta)^{\alpha}p(\theta) \tag{446}$$

Generalization in Bayesian Deep Learning

Some optimization methods (in particular, second-order batch methods) are able to find *needles in haystacks* corresponding to narrow but deep holes in the loss landscape, corresonding to parameter settings with very low loss, known as **sharp minima**.

Such solutions generally correspond to a model that has overfit the data. It is better to find points that correspond to **flat minima**, since they are more robust and generalize better. Indeed, these regions in parameter space have lots of posterior uncertainty, and hence samples from this region are less able to precisely memorize irrelevant details about the training set.

In other words, the description length for sharp minima is large, meaning you need many bits of precision to specify the exact location in parameter space to avoid uncurring large loss.

SGD often finds these flat minima by virtue of the addition of noise, therefore preventing it from entering narrow regions of the loss landscape. Also, in high dimensions, flat regions occupy a much greater volume, and hence are more discoverable by optimization procedures.

Node that a bayesian model average natively selects for flatness as the model well average weight regions with greatest volume.

Mode Connectivity and Loss landscape

DNN's have many low loss solutions, and often two independenlty trained SGD solutions can be connected by a curve in a subspace along with the training loss remains near zero. This is called **mode connectivity.**

Using bayesian model averaging, we can combine these functions togehter to provide much better performance over a single flat solution.

Effective dimensionality of a model

Modern DNN have millions of parameters, but they are often not all well-determined by the data. I.e. there can be lot of posterior uncertainty. By averaging over the posterior, we reduce the chance of overfitting, because we do not use *degrees of freedom that are not needed.*

We can quanity the number of degrees of freedom or **effective dimensionality** and define:

$$N_{eff}(H, c) = \sum_{i=1}^{k} \frac{\lambda_i}{\lambda_i + c} \tag{447}$$

where λ_i are the eigenvalues of the hessian matrix H computed at a local mode, and $c > 0$ is a regularization parameter.

Intuitively, this counts the number of well-determined parameters. A flat minimum will have many direction in parameter space that are not well determined, and hence will have low effective dimensionality.

If two models have similar training loss, but one has lower effective dimension, then it is providing a better compression of the data at the same fidelity. This leads to better generalization.

Often, depth helps provide lower effective dimensionality, leading to better compression of the data. This hierarchical inductive biases make it possible to discover more regularity in the data.

The Hypothesis space of DNN

We should distinguish the support of a model: the set of functions it can represent, from teh distribution over that support: the inductive bias which leads it to prefer some functions over others. We want to use models where the support is large, so we can capture the complexity of real-world data, but also where the inductive bias places probability mass on the kinds of functions we expect to see. If we succeed, the posterior will quickly converge on the true function after seeing a small amount of data.

Out of distribution generalization of Bayesian Neural networks

Bayesian methods are often assumed to be more robsut in the context of distribution shift, because they capture more uncertainty than methods based on point estimation.

In practice, being Bayesian only helps if we are using a good hypothesis class. If we only consider a single MLP classifier with standard gaussian priors on the weights, it is extremely unlikely that we will a compact deicions boundary because that function has neglible support under our prior. Instead we should emprace the power of Bayes to avoid overfitting and use as complex a model class as we can afford.

Online Inference

An importane application of Bayesian inference is in sequential settings, where data arrives in a continuous stream, and the model has to keep up. This is called **sequential bayesian inference** and is one approach to **online learning**.

Gaussian Processes

Introduction

Deep neural networks are a family of flexible function approximators of the form $f(x;\theta)$ where the dimensionality of θ, the number of parameters, is fixed, and independent of the size N of the training set.

Such parametric models can overfit when N is small, and underfit when N is large, due to their fixed capacity. In order to create models whose capacity automatically adapts to the amount of data, we turn to **nonparameteric models**.

We use a **gaussian process** to represent the prior $p(f)$ about our input-output mapping, we then use bayes rule to derive the posterior $p(f|\mathcal{D})$ which is another gaussian process.

Recall that a gaussian random vector of length N, $f = [f_1, .., f_n]$ is defined by its mean $\mu = \mathbb{E}[f]$ and covariance $\Sigma = Cov[f]$.

Now consider a function $f : \mathcal{X} \to \mathbb{R}$ evaluated at a set of inputs $X = \{x_n\}_{n=1}^N$ and let $f_X = [f(x_1), .., .f(x_N)]$ be the set of unknown functino values at these points. If f_X is jointly Gaussian for any set of $N \geq 1$ points, then we say that $f : \mathcal{X} \to \mathbb{R}$ is a **gaussian process**.

Such a process is defined by its **mean function** $m(x) \in \mathbb{R}$ and **covariance functino** $\mathcal{K}(x, x') \geq 0$, which is any **positive definite Mercer kernel**.

For example, we may want to use an RBF kernel of the form:

$$\mathcal{K}(x, x') \propto exp(-||x - x'||^2) \tag{448}$$

We denote the gaussian process by:

$$f(x) \sim GP(m(x), \mathcal{K}(x, x')) \tag{449}$$

where :

$$m(x) = \mathbb{E}[f(x)] \tag{450}$$

$$\mathcal{K}(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))^T] \tag{451}$$

This means that for any finite set of points $X = \{x_1, ..., x_N\}$:

$$p(f_X|\mathcal{X}) = \mathcal{N}(f_X|\mu_x, K_{x,x}) \tag{452}$$

where $\mu_X = (m(x_1), .., m(x_N))$ and $K_{X,X}(i, j) = \mathcal{K}(x_i, x_j)$

Gaussian processes can be used to define a prior over functions. We can evaluate this prior at any set of points we choose. However, to learn about the functino from data, we have to update this prior with a lieklihood function.

Typically, we assume we have a set of N iid observations $\lceil = \{(x_i, y_i) : i = 1 : N\}$ where $y_i \sim p(y|f(x_i))$.

If we use a Gaussian likelihood, we can compute the posterior $p(f|\mathcal{D})$ in closed form. FOr other kinds of likelihoods, we need to use approximate inference.

In many cases, f is not directly observed, and instead forms part of a latent variable model, both in supervised and unsupervised settings.

The generalizatino properties of a gaussian process are controlled by its covariance function or kernel. The kernels live in a **reproducting kernel hilbert space**.

Advantages of gaussian processes:

- GP provide well-calibrated predictive distributions with good characterization of epistemic (model) uncertainty - arising from not knowing which of many solutions is correct. For example, as we move away from data, there are a greater variety of consistent solutions, so we expect greater uncertainty

- GP are often state of the art for continuous regression problems, espcially spatio-temporal problems. In regression, GP inference can typically be performed in closed form
- The marginal likelihood of a GP provides a powerful mechanism for flexible kernel learning, which enable long-range extrapolations
- GP are often used a probabilistic surrogate for objectives in optimization, in a procedures known as **bayesian optimization**. To maximize an objective, we wish to move where there is a high expected value, but also to explore where we have large uncertainty. The ability for a GP to provide closed form inference in regression, in conjunction with high quality uncertainty representations, make them good here.

Mercer Kernels

The generlization properties of a GP boil down to how we encode prior knowledge about the similarity of two input vectos. If we know that x_i is similar to x_j then we can encourage the model to make the prediction oputput at bot location $f(x_i), f(x_j)$ similar.

To define similarity, we introduce the notion of a **kernel function**, which is a **positive semi definite**, a symmetric function $\mathcal{K} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$:

$$\sum_{i=1}^{N} \sim_{j=1}^{N} \mathcal{K}(x_i, x_j) c_i c_j \geq 0 \tag{453}$$

for any set of N unique points $x_i \in \mathcal{X}$ and any choice of numbers $c_i \in \mathbb{R}$.

We assume $\mathcal{K}(x_i, x_j) > 0$ so equality is achieved only if $c_i = 0$ for all i.

Equivalently, given a set of N datapoints, we can define a **gram matrix** as the following $N \times N$ similarity matrix:

$$K = \begin{matrix} \mathcal{K}(\S_\infty, \S_\infty) \cdots \mathcal{K}(\S_\infty, \S_\mathcal{N}) \\ \vdots \\ \mathcal{K}(x_N, x_1) \cdots \mathcal{K}(x_N, x_N) \end{matrix} \tag{454}$$

we say that $\mathcal{K}$ is a mercer kernel iff the gram matrix is positive definite for any set of distinct inputs $\{x_i\}_{i=1}^N$

The most widely used kernel for real-valued inputs is the **radial basis function or RBF** kernel given by:

$$\mathcal{K}(x, x'; l) = exp(-\frac{||x - x'||^2}{2l^2}) \tag{455}$$

Here l corresponds to the length-scale ofthe kernel, which is the distance over which we expect differences to matter, hence it is also called the **bandwidth parameter**.

The RBF kernel measures ismilarity between two vectors in $\mathbb{R}^d$ using scaled euclidean distance.

For real-valued inputs $\mathcal{X} = \mathbb{R}^D$, it is common to use **stationary kernels**, which are functions of the form $\mathcal{K}(x, x') = \mathcal{K}(r)$ where $r = x - x$;; thus the ouputs only depends on the relative difference between the inputs.

The **squared exponential kernel** is defined as:

$$\mathcal{K}(r; l) = exp(-\frac{r^2}{2l^2}) \tag{456}$$

where $r = ||x - x'||$

We can generalize the RBF kernel by replacing Euclidean distance with mahalnobis distance:

$$\mathcal{K}(r; \Sigma; \sigma^2) = \sigma^2 exp(-\frac{1}{2} r^T \Sigma^{-1} r) \tag{457}$$

where $= x - x'$

If Σ is diagonal, this can be simplified:

$$\mathcal{K}(r;\Sigma;\sigma^2) = \sigma^2 exp(-\frac{1}{2}r^T\Sigma^{-1}r) = \prod_{d=1}^{D}\mathcal{K}(r_d;l_d;\sigma^{\frac{2}{d}} \tag{458}$$

where

$$\mathcal{K}(r;l;\tau^2) = \tau^2 exp(-\frac{1}{2}\frac{1}{l^2}r^2) \tag{459}$$

If d is an irrelevant input dimensions we set $l_d = \infty$ so the corresponding dimension is ignored. This is known as **Automatic Relevance determination** and the kernel is called the **ARD kernel.**

Periodic Kernels

We can specify the period p to get the **periodic kernel**

$$\mathcal{K}(r;l;p) = exp(-\frac{2}{l^2}sin^2(\pi\frac{r}{p})) \tag{460}$$

where p is the period and l is the length scale.

Composition

Given two vlaid kernels $\mathcal{K}_1, \mathcal{K}_\in$, we can create a new kernel using any of the following methods:

$$\mathcal{K}(x,x') = c\mathcal{K}_1(x,x'); c > 0 \tag{461}$$
$$\mathcal{K}(x,x') = f(x)\mathcal{K})1(x,x')f(x'); functionf \tag{462}$$
$$\mathcal{K}(x,x') = q(\mathcal{K}_\infty(x,x')); polynomialq \tag{463}$$
$$\mathcal{K}(x,x') = exp(\mathcal{K}_1(x,x'))); \tag{464}$$
$$\mathcal{K}(x,x') = x^T Ax'; forpos - defmatrixA \tag{465}$$
$$\mathcal{K}(x,x') = \mathcal{K}_1(x,x') + \mathcal{K}_2(x,x') \tag{466}$$
$$\mathcal{K}(x,x') = \mathcal{K}_1(x,x') \cdot \mathcal{K}_2(x,x') \tag{467}$$

Mercers Theorems

Recall that any positive definite matrix K can be represented using an eigendecomposition of the form $K = U^T\Lambda U$ where Λ is a diagonal matrix of eigenvalues $\lambda_i > 0$ and U is a matrix containing the eigenvectors.

Now consider the (i,j) element of K:

$$k_{ij} - (\Lambda^{\frac{1}{2}U_{:i}T}(\Lambda^{\frac{1}{2}}U_{:j}) \tag{468}$$

If we define $\phi(x_i) = U_{:i}$ then we can write:

$$k_{ij} = \sum_{m=1}^{M}\lambda_m\phi_m(x_i)\phi_m(x_j) \tag{469}$$

where M is the rank of the kernel matrix.

Thus we can see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors of the kernel matrix.

This idea can be generalized to apply to kernel functions, not just kernel matrices.

We define an **eigen function** $\phi()$ of a kernel $\mathcal{K}$ with eigenvalue λ wrt the measure μ as a function that satisfies:

$$\int \mathcal{K}(x,x')\phi(x)d\mu(x) = \lambda\phi(x') \tag{470}$$

We usually sort the eigenfunctions in order of decreasing eigenvalue $\lambda_1 \geq \lambda_2 \geq ...$ The eigenfunctions are orthogonal wrt μ:

$$\int \phi_i(x)\phi_j(x)d\mu(x) = \delta_{ij} \tag{471}$$

Theorem 0.102. ***Mercer Theorem***

Any positive definite kernel function can be represented as the following sum:

$$\mathcal{K}(x, x') = \sum_{m=1}^{\infty} \lambda_m \phi_m(x)\phi_m(x;) \tag{472}$$

where ϕ_m are the eigenfunctions of the kernel, and λ_m are the corresponding eigenvalues.

*A **degenerate kernel** only has a finite number of non-zero eigenvalues, in which case we can rewrite the kernel function as an inner product between two finite length vectors.*

Thus we can replace inner product operations in an explicit possibly infinite dimensional feature space, with a call to a kernel function, replacing $\phi(x)^T\phi(x)$ with $\mathcal{K}(x, x')$.

This is called the **kernel trick**.

Kernels from Spectral Densities

Consider the case of a **shift invariant kernel** (stationary kernel) that satisfies $\mathcal{K}(x, x') = \mathcal{K}(\delta)$ with $\delta = x - x'$.

Let us further assume that $\mathcal{K}(\delta)$ is positive definite.

In this case, **Bochners Theorem** tells us that we can represent $\mathcal{K}(\delta)$ by its Fourier Transform:

$$\mathcal{K}(\delta) = \int_{\mathbb{R}^d} p(w)e^{iw^T\delta}dw \tag{473}$$

where $p(\omega)$ is the **spectral density**, a distributino over frequencies.

Random Feature Kernels

Kernelized methods take $O(N^3)$ time to invert the Gram matrix K. We can instead approximate the feature map for many kernels using a randomly chosen finite set of M basis functinos, thus reducing the cost to $O(NM + M^3)$.

GP with Gaussian Likelihood

Consider GP for regression, using a gaussian likelihood. In this case, all the computation can be performed closed form.

Suppose we observe training set $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$ where $y_n = f(x_n)$ is the noise free observation of the funciton evaluated at x_n. If we ask the GP to predict $f(x)$ for a vlaue of x already seen, we want the Gp to return $f(x)$ with no uncertainty.

In other words, it should act as an **interpolator** on the training data.

Now consider predicting outputs for new inputs not in $\mathcal{D}$.

Given a test set X_* of size $N_* \times D$ we want to predict the function outputs $f_* = [f(x_1), .., f(x_{N_*})]$ By definition of GP, the joint distribution $p(f_X, f_*|X, X_*)$ has the form:

$$\begin{pmatrix} f_X \\ f_* \end{pmatrix} \sim \mathcal{N}([\mu_X, \mu_{X_*}], [K_{X,X}, K_{X,*}, K_{X,*}^T, K_{*,*}]) \tag{474}$$

where $\mu_x = (m(x_1), ..., m(x_{N_D})$ and $\mu_* = (m(x_1^*), ..., m(x_{N_*}^*))$ and $K_{X,X} = \mathcal{K}(X, X)$ is $N_D \times N_D$, $K_{X,*} = \mathcal{K}(X, X_*)$ is $N_D \times N_*$ and $K_{*,*} = \mathcal{K}(X_*, X_*)$ is $N_* \times N_*$

By Gaussian rules, the posterior has the form:

$$p(f_*|X_*, \mathcal{D}) = \mathcal{N}(f_*|\mu_{*|X}, \Sigma_{*|X} \tag{475}$$

$$\mu_{*|X} = \mu_* + K_{X,*}^T K_{X,X}^{-1}(f_X - \mu_X) \tag{476}$$

$$\Sigma_{*|X} = K_{*,*} - K_{X,*}^T K^{-1} K_{X,*} \tag{477}$$

Now let use drop the assumption of noiseless observation, and put $y_n = f(x_n) + \varepsilon_n$ where $\varepsilon_n \sim \mathcal{N}(0, \sigma_y^2)$.

The covariance of the observed noisy responses is:

$$Cov[y_i, y_j] = Cov[f_i, f_j] + Cov[\varepsilon_i, \varepsilon_j] = \mathcal{K}(x_i, x_j) + \sigma_y^2 \delta_{ij} \tag{478}$$

In other words:

$$Cov[y|X] = K_{X,X} + \sigma_y^2 I_N \tag{479}$$

The joint density now looks like:

$$\begin{pmatrix} y \\ f_* \end{pmatrix} \sim \mathcal{N}([\mu_X, \mu_{X_*}], [K_{X,X} + \sigma_y^2 I, K_{X,*}, K_{X,*}^T, K_{*,*}]) \tag{480}$$

and the posterior predictive density at a set of test points X_* is:

$$p(f_*|X_*, \mathcal{D}) = \mathcal{N}(f_*|\mu_{*|X}, \Sigma_{*|X} \tag{481}$$

$$\mu_{*|X} = \mu_* + K_{X,*}^T (K_{X,X} + \sigma_y^2 I)^{-1}(y - \mu_X) \tag{482}$$

$$\Sigma_{*|X} = K_{*,*} - K_{X,*}^T (K_{X,X} + \sigma_y^2 I)^{-1} K_{X,*} \tag{483}$$

This amounts to writing a linear combination (for a single test point):

$$\sum_{n=1}^{N} \mathcal{K}(x_*, x_n)\alpha_n \tag{484}$$

where

$$K_\sigma = K_{X,X} + \sigma_y^2 I \tag{485}$$

$$\alpha = K_\sigma^{-1} y \tag{486}$$

when the mean function is zero.

We can compute the Cholesky decomposition of K_σ, and once we have computed α, we can compute predictions for each test point in $O(N)$ time for the mean, and $O(N^2)$ time for the variance.

Note: bayesian linear regression is a special case of a GP, which assume that the feature $\phi(x)$ are finite length vector, whereas in GP, we can work directly in terms of kernels, which may correspond to infinite length feature vectors, and hence work in **function space**.

Kernel Ridge Regression

Ridge regression refers to linear regression with a l_2 penalty on the weights:

$$w^* = argmin_w \sum_{n=1}^{N} (y_n - f(x_n; w))^2 + \lambda ||w||^2 \tag{487}$$

where $f(x; w) = w^T x$

The solution is:

$$w^* = (X^T x + \lambda I)^{-1} X^T y \tag{488}$$

We can consider a function space version of this:

$$f^* = argmin_{f \in \mathcal{F}} \sum_{n=1}^{N} (f_n - f(x_n))^2 + \lambda ||f||^2 \tag{489}$$

Let $\mathcal{F} : \{f : \mathcal{X} \to \mathbb{R}\}$ be a hilbert space $\mathcal{H}$

which is a complete inner product space.

A **Reproducing Kernel Hilbert space** is defined as follows.

Let $\mathcal{H}$ be a hilbert space of real valued functions. We say that $\mathcal{H}$ is a *reproducing kernel hilbert space with inner product* $\langle , \rangle_H$ if there exist s asymmetric kernel function $\mathcal{K}L\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ s.t.:

- For every $x \in \mathcal{X}$ $\mathcal{K}(x, \cdot) \in \mathcal{H}$
- $\mathcal{K}$ is reproducable:

$$\langle f(\cdot), \mathcal{K}(\cdot, x') \rangle = f(x') \tag{490}$$

The utility of RKHS in ml is it allows us to define a notion of smoothness or complexity in terms of norm.

Indeed, we can put:

$$||f||_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{\infty} \frac{f_i^2}{\lambda_i} \tag{491}$$

Theorem 0.103. ***Representer Theorem***

If we want to solve regularized empirical risk minimization in function space:

$$f^* = argmin_{f \in \mathcal{H}_\mathcal{K}} \sum_{n=1}^{N} l(y_n, f(x_n)) + \frac{\lambda}{2} ||f||_{\mathcal{H}}^2 \tag{492}$$

where $\mathcal{H}_\mathcal{K}$ is a RKHS with kernel $\mathcal{K}$ and l is a loss function, then:

$$f^*(x) = \sum_{n=1}^{N} \alpha_n \mathcal{K}(x, x_n) \tag{493}$$

where α_n are coefficients that depend on the training data.

GP with non-gaussian likelihoods

Need to approximate

Scaling GP Inference to Large Datasets

We saw that the best to peform GP inefrence is to compute Cholesky decomposition of the N by N gram matrix, but this takes $O(N^3)$ time.

Subset of the Data

If we sample M example from out dataset, exact inference takes $O(M^3)$ time.

We want our sampled subset to cover the original data, so it contains approxiamtely the same information. One method is to use clustering.

Nystrom Approxination

Suppose we had a rank M approximation to the $N \times N$ gram amtrix of the form:

$$K_{X,X} \approx U \Lambda U^T \tag{494}$$

where Λ is a diagonal matrix of the M leading eigenvalues, and U is teh matrix of the corresponding M eigenvectors, each of size N.

By the **matrix inversion lemma**, we can approximately compute the inverse in $O(NM^2)$ time.

Eigendecomposition however, still takes $O(N^3)$ time, which does not help. By partitioning the matrix we can perform a **Nystrom approximation**.

Inducing Point Methods

Sparse Variational Methods

Exploiting Parallelization and structure via kernel matrix multiplies

Learning the kernel

GP and DNN

Theorem 0.104. *An MLP with one hidden layer whose width goes to infinity and which has a gaussian prior on all the parameters converges to a Gaussian process with a well defined kernel.*

Gaussian processes for timeseries forecasting

WE can use GP to peform timeseries forecasting by modelling the unknown output as a function of time $f(t)$ and to represent aprior about the form of f as a GP; we than update this prior given evidence, and forecast into the future.

Naively this would take $O(T^3)$ time, but with stationary kernels, it is possible to reformulate the problem as a linear-gaussian state space model, and then use kalman smoother to perform inference in linear time.

Beyond IID Assumption

Distribution Shift

Suppose we have a labeled training set from a **source distribution** $p(x, y)$ which we use to fit a predictive model $p(y|x)$. At test time we encounter data from the **target distribution** $q(x, y)$.

If $p \neq q$ we say that there is **distribution shift**.

The four main types of distribution shift

1. **Covariate shift**: If a causal discriminative model changes such that $p_\psi(x)$ changes with $\psi_s \neq \psi_t$ this is covariate or domain shift. For example, the training set may be clean images of coffee pots, at test time they have gaussian noise
2. **Concept shift**: in a causal discriminative model if $p_w(y|x)$ changes so that $w_s \neq w_t$ this is called concept shift or anotation shift. For example, the labelling process changed at test time compared to training, may include new categories etc.
3. **Label prior shift**: in a anti-causal generative model if $p_{\pi(y)}$ changes so that $\pi_s \neq \pi_t$ we call it label shift or prior shift. For example, in medical images, $Y = 1$ implies disease, then an urban hospital in the training set and rural hospital in test set implies that the prevalence of the disease represented by $p(Y = 1)$ is very different
4. **Manifestation shift**: in a anti-causal generative model if $p_\phi(x|y)$ changes so that $\phi_s \neq \phi_t$ this is manifestation shift or conditional shift. For example, the way that the same disease Y manifests itself in the shape of a tumor X might be different. This is usually the presence of hidden confounding factors that were changed between the source and the target.

Detecting Distribution Shift

Detecting shifts using two sample testing

Suppose we can collect a set of samples from the source and target distribution. We can then use standard techniques for two sampling testing to estimate if the nul hypothesis $p(x, y) = q(x, y)$ is true or not.

Detecting single out of distribution inputs

Now suppose we have *one* unlabelled sample form the target distribution $x \sim q$ and we want to know if x is in -distribution or out of distribution.

Anomaly detection requires making a binary decision about whether the test sample is in or out, we can fit a binary classifier to distinguish these. We can also use the classifier logits to derive a **confidence score** or **uncertainty metric**.

Robustness to Distribution Shift

Data augmentation

A simple approach is to simulate samples from the target distribution by modifying the source data.

For example, apply small perturbation in images while keeping the label the same.

Adapting to Distribution Shift

Supervised adapatation using transfer learning

Suppose we have labeled training data from a source distribution $\mathcal{D}^s = \{(x_n, y_n) \sim p : n = 1, ..., N_s\}$ and also some labeled data from the target distribution $\mathcal{D}^t = \{(x_n, y_n) \sim q : n = 1, .., N_t\}$. Our goal is to minimize the empirical risk on the target q:

$$\hat{R}(f, \mathcal{D}^t) = \frac{1}{|\mathcal{D}^t|} \sum_{(x_n, y_n) \in \mathcal{D}^t} l(y_n, f(x_n)) \tag{495}$$

If the test set is large, we can optimize directly using standard empirical risk minimization. But if the test set is small, we might want to use $\mathcal{D}^s$ as a regularizer. This is called **transfer learning**

For example, we can **pre-train and fine tune**. Since we assume we have few samples at test set, we can freeze many of the learned pre-trained parameters from the source model.

Unsupervised domain adaptation for covariate shift

This is a method when we only need access to unlabeled examples from the target distribution.

One method called **domain adversarial learning** involves learning a feature space so that it cannot distinguish whether the input is coming from the source or target distribution.

Learning from Multiple Distributions

Multi task learning

In **multi task learning** we have labeled data from J different distributions $\mathcal{D}^J = \{(x_n^j, y_n^j) : n = 1 : N_j\}$ and the goal is to learn a model that predicts well on all J of them simultaneously where $f(x, j) : \mathcal{X} \to \mathcal{Y}_j$ is the output for the jth task.

The simplest approach is to fit a single model with multiple output heads, this is called a **shared trunk network**.

Another approach, is the take a weighted combination of teh activations of each single task network, called **cross stish networks.**

In cases where there is *task interfereence* or **negative trasnfer**, multi task learning may seriously degrade performance, and we should just use separate networks.

Domain generalization

In **domain generalization**, we have labelled data from J different distributions, and then we seek to learn on J+1 new distribution at test time.

Typically there is some associated *meta data or context* shared between each environment.

Continual Learning

In **continual learning** or **life long learning** a system learns from a sequence of different distribution $p_1, p_2, ...$

So that at each step the model receives a new batch of labelled data.

In **domain drift**, the underlying prior distribution $p_t(x)$ changes over time (covariate shift) but the functional mapping $f_t : \mathcal{X} \to \mathcal{Y}$ is constant over time.

In **concept drift**, the functional mapping $f_t : \mathcal{X} \to \mathcal{Y}$ changes over time, but the input distribution $p_t(x)$ is constant.

Online learning

In online learning, the agent sees one new example, and must predict the right output, given all the prior training data. We evaluate on this example alone, rather than a fixed test set.

Often, we formulate this in terms of **regret**, which is to compare it to an optimal value that could have been obtained in hindsight:

$$regret = \sum_{t=1}^{T} [l(\hat{p}_{t|t-1}, y_t - l(\hat{p}_{t|T}, y_t)] \tag{496}$$

where the first term is the online prediction, and the second is the optimal estaimte at the end of training. It is possible to convert bounds on regret, which are backwards looking into bounds on risk (expected future loss) which is forward looking.

Adversarial examples

Gradient Based Attacks

We find a small perturbation δ to add to the input x to create $x_{adv} = x + \delta$ so that $f(x_{adv}) = y'$ and $y =\neq y$. This is known as a **target attack**. We can also choose our adversarial input such the new input is just different $f(x + \delta) \neq f(x)$ which is called an **untargeted attack**.

To define what we mean by *small perturbation*, we impose the constraint $x_{adv} \in \mathcal{B}_\varepsilon(x) = \{x' : ||x' - x||_p < \varepsilon\}$.

A *whitebox attack* occurs when the attacker knows the model parameters θ.

This lets us use gradient based optimization with constraints, such as bound-constrained gradient descent.

Black box (gradient free) attacks

If we no longer assume that the adversaery knows the parameters θ of the predictive model we must use derivative free optimization such as evolutionary algorithms.

The existence of adverserial examples in high dimensional spaces is due to **concentration of measure** and is a property of many high dimensional data distributions.

Generation

Introduction

A **generative model** is a joint probability distribution $p(x), x \in \mathcal{X}$.

In some cases, the model may be conditioned on inputs or covariates $c \in \mathcal{C}$ which gives rise to **conditional generative models** of the form $p(x|c)$.

At a high level, there are **deep generative models** which use a deep neural network to learn a complex mapping from a single latent vector z to the observed data x, vs. **probabilistic graphical models** that map a set of interconnected latent variabels $z_1, .., z_L$ to the observed variables $x_1, ..., , x_D$ using often simpler, linear mappings.

The 4 main contains deep generative models are **variational autoencoders, autoregressive models, normalizing flows, diffusion models, energy based models, and Gans**.

We categorize these models in terms of the following criteria:

- Density: does the model support pointwise evaluation of the probability density function $p(x)$ and if so, is this fast or slow, exact or approximate, or a bound etc. For **implicit models** such as GANS, there is no well-defined density $p(x)$. For other models, we can only compute a lower bound on the density (VAE)
- Sampling: does the model support generating new samples $x \sim p(x)$ and if so, is this fast or slow, exact or approximate? Directed PGM, VAE, and GANS all support fast sampling, however PGM, energy based models, autoregressive models, diffusion and flows are slow for sampling
- Training: what kind of method is used for parameter estimation? For some models like autoregressive or flows, we can perform exact MLE although the objective is usually non-convex, so we can only reach a local optima. For other models, we cannot tractably compute the likelihood, so for VAE we maximize a lower bound on the likelihoood. or Gans, we use min-max training, which can be unstable
- Latents: does the model use a latent vector z to generate x or not, and if so, is it the same size as x or is it a potentially compressed representation?
- Architecture; what kind of neural network should we use, and are there restrictions? For flows, we are restricted to using invertible neural networks where each layer has a tractable Jacobian.

Goals of Generative Modelling

- Generating data: create new data samples. Often it is useful to use a conditional generative model of the form $p(x|c)$
- Density estimation: evaluting the probability of an observed data vector (computing $p(x)$ which can be used for outlier detection, data compression, generativew classifiers etc.
- Imputation: filling in missing values of a data vector or data matrix
- Structure discovery: use bayes rule to invert the model given latent factors that we assume are the causes that generated the observed data x
- Latent space interpolation: interpolate desired properties, by using the fact that taking linear interpolation works since the learned manifold often has approximately zero curvature. We can also do latent space arithmetic under the same justification.

Evaluating Generative models

We require metrics that capture **sample quality** and **sample diversity** and **generalization**

In **likelihood based evaluation** we look at the KL divergence between the model q and the true distribution p:

$$D_{KL}(p|q) = \int p(x) log \frac{p(x)}{q(x)} = -\mathbf{H}(p) + \mathbf{H}_{ce}(p, q) \tag{497}$$

Since the entropy is constant, we can go ahead and approximate $p(x)$ by the empirical distribution, and evalute the cross entropy in terms of the **empirical negative log likelihood on the dataset**:

$$NLL = -\frac{1}{N} \sum_{n=1}^{N} log q(x_n) \tag{498}$$

For models of discrete data, such as language models, it is easy to compute the negative log likelihood, however it is common to measure performance using a quanity called **perplexity**.

Unfortunetely, for many models, it is intractable to compute, or non-existant (GAN). Also, empirically, the likelihood does not relate well to sample quality.

Indeed, in high dimensional spaces, a mixture of good classifier and noise will look bad, but have negligble drop in likelihood for high dimensions. Conversely, a noisy average of training data will look great but have bad likelihood on test due to overfitting.

We often use distances and divergences in feature space.

For example the **inception score** measure the KL divergence between teh marginal distribution of class labels obtained from the samples and the distribution obtained from a true sample:

$$InceptionScore = exp[\mathbf{E}_{p_\theta(x)} D_{KL}(p(y|x)|p_\theta(x))]] \tag{499}$$

The inception score solely relies on class labels, and thus does not measure overfitting or sample diversity outside the predifined dataset classes. For example, a model that generates one perfect example per class would get a perfect inception score, despite not capturing the variety of examples.

To address this, the **Frechet inception distance** measures the frechet distance between two gaussian distributions on sets of features of a pre-trained classifer. One gaussian is obtained by passing the model samples through a pretrained classifier, and the other by passing samples from the dataset through the same classifier.

If the mean and covariance obtained from model features are μ_m, Σ_m and those from the data are μ_d, Σ_d then the *frechet inception distance* is:

$$FID = ||\mu_m - \mu_d||_2^2 + trace(\Sigma_d + \Sigma_m - 2(\Sigma_d \Sigma_m)^{\frac{1}{2}}) \tag{500}$$

Variational AutoEncoders

Variational Autoencoders are generative models of the form:

$$z \sim p_\theta(z) \tag{501}$$
$$x|z \sim ExpFam(x|d_\theta(z)) \tag{502}$$

where $p(z)$ is some kind of prior on the latent code z, $d_\theta(z)$ is a deep neurla network, known as the **decoder** and $Expfam(x|\eta)$ is an exponential family distribution.

This setup is called a **deep latent variable model**.

Computing the posterior $p_\theta(z|x)$ is inctractable, as is computing the marginal likelihood:

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz \tag{503}$$

hence we resort to approximate infernce.

Typically we use **amortized inference** which involves training another model $q_\phi(z|x)$ called the **recognition network** simultaneously with the generative model to do approximate posterior inference.

Together, this is the **variational autoencoder**.

VAE Basics

In the simplest setting, the VAE defines a generative model of the form:

$$p_\theta(z, x) = p_\theta(z)p_\theta(x|z) \tag{504}$$

where $p_\theta(z)$ is usually gaussian and $p_\theta(x|z)$ is a product of exponential family distribution with parameters computed by a neural network decoder $d_\theta(z)$.

In addition, a VAE fits a recognition network:

$$q_\phi(z|x) \approx p_\theta(z|x) \tag{505}$$

to perform approximate posterior inference.

Here $q_\phi(z|x)$ is usually gaussian with parameters computed by a neural network **encoder** $e_\phi(x)$

$$q_\phi(z|x) = \mathcal{N}(z|\mu, diag(exp(l)) \tag{506}$$
$$(\mu, l) = e_\phi(x) \tag{507}$$

where $l = log\sigma$.

The model encodes the input x into a stochastic latent bottleneck z and then decoding it to approximately reconstruct the input.

The idea of training an inference network to "invert" a generative network rather than running an optimziation algorithm to infer the latent code is called **amortized inference.**

The VAE optimizes a variational lower bound on the log likelihood which means that convergence to a locally optimal MLE of the parameters is guarenteed.

Evidence lower bound

When fitting the model, our goal is to maximize the marginal likelihood

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz \tag{508}$$

This is intractable, but we can use the inference network to approxaimte the posterior $q_\phi(z|x)$ and hence a lower bound to the marginal likelihood.

Note that:

$$logp_\theta(x) = \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x)] = \mathbb{E}_{q_\phi(z|x)}[log(\frac{p_\theta(x,z)}{q_\phi(z|x)})] + \mathbb{E}_{q_\phi(z|x)}[log(\frac{q_\phi(z|x)}{p_\theta(z|x)})] = ELBO_{\phi,\theta}(x) + D_{KL}(q_\phi(z|x)|p_\theta(z|x)) \tag{509}$$

and hence $ELBO_{\phi,\theta}(x) \leq logp_\theta(x)$, so that the elbo is a lower bound on the log marginal likelihood or **evidence**.

There are three equivalent ways to write the elbo:

$$\begin{aligned} ELBO_{\theta,\phi}(x) &= \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x,z) - logq_\phi(z|x)] &(510)\\ &= \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z) + logp_\theta(z)] + \mathbb{H}(q_\phi(z|x)) &(511)\\ &= \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z) - D_{KL}(q_\phi(z|x)|p_\theta(z|x)) &(512)\end{aligned}$$

In words:

1. When sampling from the recognition model, maximize the marginal
2. When sampling from the recognition model, maximize the conditional with maximal entropy
3. When sampling from the recognition model, maximize the conditional and minimize divergence from our prior on latents

A better approximation to the posterior results in a tighter bound. When the KL goes to zero in the final definition, the posteior is exact, so any improvements to the ELBO directional translate to improvements in the likelihood of the data, as in EM.

The negative ELBO is called the **variational free energy.**

We can approximate the ELBO by sampling from the posterior $z_s \sim q_\phi(z|x)$ and then evaluating use M.C:

$$\mathbb{E}_{q_\phi(z|x)}[logp_\theta(x,z)] \sim \frac{1}{S}\sum_{s=1}^{S} logp_\theta(x,z_s) \tag{513}$$

and the we can compute the differential entropy term $\mathbb{H}[q_\phi(z|x)]$ analytically based on choice of variational distribution.

Alternatively, we can monte carlo sample all terms:

$$ELBO_{\theta,\phi}(x) \sim \frac{1}{S}\sum_{s=1}^{S} logp_\theta(x|z_s) - +logp_\theta(z_s) - logq_\phi(z_s|x)] \tag{514}$$

$$\tag{515}$$

Note that θ fits the model to the data, and ϕ reduces the KL gap between the approximate and true posterior.

The gradient wrt the generative parameters θ are easy to compute, since we can push gradients inside the expectation and use a single MC sample:

$$\begin{aligned} \nabla_\theta ELBO_{\theta,\phi}(x) &= \nabla_\theta \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x,z) - logq_\phi(z|x)] &(516)\\ &= \mathbb{E}_{q_\phi(z|x)}[\nabla_\theta\{logp_\theta(x,z) - logq_\phi(z|x)]\} &(517)\\ &\approx \nabla_\theta logp_\theta(x,z_s) &(518)\end{aligned}$$

The is unbiased estimator so can be used with SGD.

The gradient wrt to the inference parameters ϕ is more difficult since we cannot push the gradient inside. So we must use the **reparameterization trick to compute ELBO gradients**.

We rewrite $z \sim q_\phi(z|x)$ as some invertible differntiable transformation r of another random variable $\varepsilon \sim p(\varepsilon)$ which does not depend on ϕ:

$$z \sim \mathcal{N}(\mu, diag(\sigma)) \iff z = \mu + \varepsilon \cdot \sigma, \varepsilon \sim \mathcal{N}(0, I) \tag{519}$$

and thus:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[logp_\theta(x,z) - logq_\phi(z|x)] = \mathbb{E}_{p(\varepsilon)}[\nabla_\phi\{logp_\theta(x,z) - logq_\phi(z|x)\}] \tag{520}$$

which we can approximate with a single MC sample.

Since we are working with a new r.v ε, we need to sue the change of variables formula to compute:

$$logq_\phi(z|x) = logp(\varepsilon) - log|det(\frac{\partial z}{\partial \varepsilon})| \tag{521}$$

and we design the transformation such that this Jacobian is easy to compute.

For example, in a fully factorized gaussian, the Jacobian is $diag(\sigma)$.

VAE Generalizations

β-VAE

It is often that VAE generate somewhat blurry images. This is because for a fixed inference network, the optimal setting of generator parameters when using squared reconstruciton loss is to ensure the decoder predicts the average of all inputs x that map to a given latent code z, hence blurry.

We solve this by increasing the expresive power of the posterior approxiamtion (avoiding the merging of distinct inputs into the same latent code) or of the generator (by adding back informaiton that is missing from the latent code).

A simple solution is to reduce the penalty on the KL term, making the model closer to a determinsitc auto encoder:

$$\mathcal{L}_\beta(\theta, \phi|x) = -\mathbb{E}_{q_\phi(z|x)}[logp_\theta(x|z)] + \beta D_{KL}(q_\phi(z|x)|p_\theta(z)) \tag{522}$$

where the first term is the reconstruction error (negative log likelihood.

This is the β **VAE objective**. If we set $\beta = 1$ we recover the standard VAE, and if $\beta = 0$ then we recover the objective used in standard autoencoders. If we use $\beta > 1$ is encourages the learning of a latent representation that is **disentangled**. This means that each altent dimension represents a different factor of variation of the input.

Note: the $\beta - VAE$ is an unsupervised version of the **informaiton bottelenck objective**.

Info-VAE

Some drawbacks of the standard ELBO objective is the tendency to ignore the latent code when the decoder is powerful, and the tendency to learn a poor posterior approximation due to the mismatch between the KL terms in data space and in latent space.

We can fix this by using a generaliation objective called **INFO-VAE**:

$$ELBO(\theta, \phi|x) = -\lambda D_{KL}(q_\phi(z)|p_\theta(z)) - \mathbb{E}_{q_{\phi(z)}}[D_{KL}(q_\phi(x|z)|p_\theta(x|z)] + \alpha \mathbb{I}_q(x,z) \tag{523}$$

where $\alpha \geq 0$ controls how much we weight the mutual information between x and z, and $\lambda \geq 0$ controls the tradeoff between z-space KL and x-space KL.

With $\alpha = 0, \lambda = 1$ we get standard ELBO.

If we set $\alpha = 1$, we recover the **MMD-VAE**, which replaces the KL divergence with the maximum mean discrepancy divergence.

Multi-model VAEs

It is possible to extend VAE to create oint distributions over different kind of variables like images and text, this is called the **Multi-modal VAE**. We assume the M modalities are conditionally independent given the latent code, and hence:

$$p_\theta(x_1, ..., x_M, z) = p(z) \prod_{m=1}^{M} p_\theta(x_m|z) \tag{524}$$

VAEs with sequential encoders / decoders

We assume the data x is a variable length sequence, but we have a fixed sized latent variable $z \in \mathbb{R}^K$.

If we use RNN for the encoder and decoder of a VAE we get a mdoel called a **VAE-RNN**.

The generative model is $p(z, x_{1:T}) = p9z)RNN(x_{1:T}|z)$, the inference model is $q(z|x_{1:T}) = \mathcal{N}(z|\mu(h), \Sigma(h))$, where h is the output of a bidirectional RNN applied to $x_{1:T}$.

Avoiding Posterior Collapse

If the decoder $p_\theta(x|z)$ is sufficiently powerful, then the VAE does not need to use the latent code for anything. This is called **posterior collapse**.

Solutions

- KL annealing: scale KL penalty in ELBO by β from 0 (autoencoder) to 1 (mle training)
- Lower bounding the rate: **free bits** lower bounds the rate by replacing the KL loss with a hinge loss, that will give up driving down the KL for dimensions that are already beneatha target compression rate λ.
- Adding skip connections (**Skip-VAE**)
- Improved variational inference: the posterior collapse is partly a result of a poor approximation to the posterior. Here, we more aggressively update the inference network before each step of generative model fitting.
- Use Info-vae or MMD-autoencoder objectives instead of vanilla ELBO. The Info-VAE objective includes a term to explicitly enforce a non-zero mutual informaiton between x and z.

VAE with hierarchical structure

A **hierarchical VAE** with L stochastic layers is the following generative model:

$$p_\theta(x, z_{1:L} = p_\theta(z_L)[\prod_{l=L-1}^{1} p_\theta(z_l|z_{l+1})]p_\theta(x|z_1) \tag{525}$$

We can improve on the above model by making it non-Markovian, leeting each z_l depend on all the higher level stochastic variables, not just the preceding level:

$$p_\theta(x, z) = p_\theta(z_L)[\prod_{L-1}^{1} p_\theta(z_l|z_{l+1:L})]p_\theta(x|z_{1:L} \tag{526}$$

To perform inference in a hierarhical VAE, we oculd use a **bottom-up inference model** of the form:

$$q_\phi(z|x) = q_\phi(z_1|x)\prod_{l=2}^{L} q_\phi(z_l|x, z_{1:l-1}) \tag{527}$$

However a better approach is to use a **top down inference model** of the form

$$q_\phi(z|x) = q_\phi(z_L|x)\prod_{l=L-1} 1q_\phi(z_l|x, z_{l+1:L}) \tag{528}$$

The top down inference model is better because it more closely approximates the true posterior of a given layer, which is given by:

$$p_\theta(z_l, x, z_{l+1:L} \propto p_\theta(z_l|z_{l+1:L}p_\theta(x|z_l, z_{l+1:L}) \tag{529}$$

Very Deep VAE

State of the art hierarchical vae model.

The architecture is a simple convolutional VAE with bidirecitonal inference. For each layer, the prior an posterior are diagonal Gaussians.

Nearest neighbour upsampling in the decoder worked much better than transposed convolution, and avoided posterior collapse.

Variational pruning is when higher level latent layers are often ignored, so the model does not learn interesting high level semantics. A common heuristic to mitigate this problem is to use KL balancing coefficients to ensure than an equal amount of information is encoded in each layer.

Vector quantization VAE

The **VQ-VAE** is a vector quanitized VAE, which is like a standard VAE except it uses a set of discrete latent variables.

Autoencoder with binary code

The simplest approach to the problem is to construct a standard VAE but to add a discretization layer at the end of teh encoder $z_e(x) \in \{0, ..., S-1\}^K$ where S is the number of states and K is the number of discrete latents.

The discontinuous quantization operation of the encoder prohibits the direct use of gradient based optimization.

We can use a straight through estimator.

VQ-VAE Model

We can get a more expressive model by using a 3d tensor of discrete latents $z \in \mathbb{R}^{H\times W\times K}$ where K is the number of discrete values per latent variable. Rather than binarizing the continuous vector $z_e(x)_{ij}$ we comapre it to a **codeback** of embedding vectors $\{e_k : k = 1 : K, e_k \in \mathbb{R}^L\}$ and then set z_{ij} to the index of the nearest codebook entry:

$$q(z_{ij} = k|x) = \begin{cases} 1 & if k = argmin_{k'}||z_e(x)_{ij} - e_{k'}||_2 \\ 0 & otherwise \end{cases} \tag{530}$$

When reconstructing the input, we replace each discrete code index by the corresponding real-valued codebook vector:

$$(z_q)_{ij} = e_k, z_{ij} = k \tag{531}$$

These values are then passed to the decoder $p(x|z_q)$ as usual.

We can fit this model by minimizing the negative log likelihood (reconstruction error) using the straight through estimator, which amounts to passing the gradients from the decoder input $z_q(x)$ to the encoder $z_e(x)$.

This would mean the codebook entries will not get any learning signal, so we add an extra term in the loss, the **codebook loss** that encourages the codebook entries e to match the output of the encoder.

We treat the encoder $z_e(x)$ as a fixed target, by assing a **stop gradient** operator on it; this ensures z_e is treated normally in the forwards pass, but has zero gradient in the backwards pass.

The modifies loss becomes:

$$\mathcal{L} = -logp(x|z_q(x)) + ||sg(z_e(x)) - e||_2^2 \tag{532}$$

where e refers to the codebook vector assigned to $z_e(x)$ and sg is the stop gradient operator.

This procedure will learn to update the codeboook vectors so it matcehs the output of the encoder. It is also important to ensure the encoder does not change its mind, too often what codebook value to use.

To prevent this, the authors propose to add a third term to the loss known as the **commitement loss**, that encourages the encoder output to be close to the codebook values. Thus we get the final loss:

$$\mathcal{L} = -logp(x|z_q(x)) + ||sg(z_e(x)) - e||_2^2 + \beta||z_e(x) - sg(e)||_2^2 \tag{533}$$

Overall, the decoder optimizes the first term only, the encoder optimizes the first and last term, and the embeddings optimize the middle term.

Discrete VAE

In VQ-VAE we use a one-hot encoding for the latents $q9z = k|x) = 1$ iff $k = argmin_k||z_e(x) - e_k||_2$ and then set $z - q = e_k$. This does not capture uncertainty in the latent code and requires the use of the straight through estimator for training.

In DALL-E paper, they use a faily simple method, based on using the Gumbel-softmax relaxation for discrete variables. In brief let $q(z = k|x)$ be the probability that the input x is assigned to codebook entry k.

We can exactly sample $w_k \sim q(z = k|x)$ form this by computing $w_k = argmax_k g_k + logq(z = k|x)$ where each g_k is from a gumbel distribution. We can now relax this by using a softmax with termperature $\tau > 0$ and computing:

$$w_k = \frac{exp(\frac{g_k + logq(z=k|x)}{\tau})}{\sum_{j=1}^{K} \frac{exp(g_j + logq(z=j|x))}{\tau}} \tag{534}$$

We now set the latent code to be a weighted sum of the codebook vectors:

$$z_q = \sum_{k=1}^{K} w_k e_k \tag{535}$$

In the limit $\tau \to 0$ the distribution over weights w converges to a one hot distribution, in which case z becomes equal to one of the codebook entries.

Now the ELBO is differentiable as usual. Unlike VQ-VAE the KL term is not a constant because the encoder is stochastic. Furthermore since the Gumbel noise variables are sampled from a distribution that is independent of the encoder parameters, we can use the reparameterization trick to optimize.

VG-Gan

One drawback of VQ-VAE is that is uses mean squared error in its reconstruction loss, which can result in blurry samples. In VQ-GAN they replace this with a patch-wise GAN loss, together with a perceptual loss, and use a transformer to model the prior on th elatent codes. They call the model with trasnformers **VIM: Vector quantized image modelling**

Auto-Regressive Models

By the chain rule of probability, we can write any joint distribution over T variables as:

$$p(x_1 : T) = \prod_{t=1}^{T} p(x_t|x_{1:t-1}) \tag{536}$$

where $p(x_1|x_{1:0}) = p(x_1)$.

This is called an **autoregressive model**, which coresspdons to a fully connected DAG.

Thde models can b conditioned on arbitrary inputs or context c, in order to define $p(x|c)$.

For tractbility, we can make the model a **first order markov assumption markov model** which simplifies $p(x_t|x_{1:t-1}) = p(x_t|x_{t-1})$.

This is restricting, so we can make x_t depend on all of the past $x_{1:t-1}$ without explicitly regressing on them, and assume the past can be compresed into a **hidden state** z_t.

Neural AutogRegressive density estimators (NADE)

A simple way to represent each conditional distribution by a generalize linear model. But we can instead fit a neural network.

Instead of using T seperate neural networks, it is more efficient to create a single network with T inputs and T outputs, and use masking. This is called **Masked autoencoder for density estimation**.

1-d Causal CNN Example: Wavenet

Example: using **dilated convolution** to capture long term dependenceis, the **wavenet** model is a state of the art **text to speech** system, which is a conditional autoregresisve model $p(x|c)$ where c is a set of linguistic features derived from an input sequences of words, and x is araw audio.

2-d Causal CNN Example: Pixel-CNN

We can extend causla convolution to 2d, to get an autoregressive model:

$$p(x|\theta) = \prod_{r=1}^{R} \prod_{c=1}^{C} p(x_{r,c}|f_\theta(x_{1:r-1,1:C}, x_{r,1:c-1})) \tag{537}$$

where R,C is the number of rows and columns, and we condition on all previously generated pixels in **raster scan order**.

Transformer Decoders

We introduced trasnformers for encoder sequences (BERT). They can also be used for decoding (generating) sequences.

At each step t the model applies masked self-attention to the first t inputs $y_{1:t}$ to compute a set of attention weights $a_{1:t}$. We use these to compute an activation vector $z_t = \langle a_t, y_t \rangle$. This is passed through a feed-forward layer to compute $h_t = MLP(z_t)$. This is repeated on multiple layer, and finally the output is used to predict the next element in the sequence $y_{t+1} \sim Cat(softmax(Wh_t))$.

At training, all predictions can happen in parallel since the target generated sequence is already available. (next token prediction).

However at test time, the model be applied sequentially. Note the running time of transformers is $O(T^2)$ naively.

GPT

Generative Pre-training transformer is a decoder-only transformer model that uses masked attention.

DALL-E

The DALL-E model is a **text-to-image** model that can generate images conditions on text.

The basic idea is to transform an image x into a sequence of discrete tokens z using a discrete VAE model which defines the joint $p(x, z)$. We then fit a transformer to the concatenation of the image tokens z and text tokens y to get a model of the form $p(z, y)$.

To sample an image x given a text prompt y, we sample a latent code $z \sim p(z|y)$ and then we feed z into a VAE decoder to get $x \sim p(x|z)$.

Multiples images are generated for each prompt, and these are ranked according to pre-trained critic, which gives them scores depending on how well the generated images matches the input text $s_n = critic(x_n, y_n)$.

The critic they used was the contrasting CLIP model.

Normalizing Flows

Normalizing flows is a class of flexible density models that can be easily sampled from and whose exact likelihood function is efficient to compute.

Normalizing flows create complex distributions $p(x)$ by passing random variables $u \in \mathbb{R}^D$ drawn from a simple **base distribution** $p(u)$ through a nonlinear but *invertible* transformation $f : \mathbb{R}^D \to \mathbb{R}^D$:

$$x = f(u) \tag{538}$$

$$u \sim p(u) \tag{539}$$

The base distribution is typically Gaussian or uniform. Sampling is easy, we sample $u \sim p(u)$ and the compute $x = f(u)$.

To compute the density $p(x)$ we rely on the fact that f is invertible. Let $g(x) = f^{-1}(x) = u$ be the inverse mapping that **normalizes** the data distribution by mapping it back ot teh base distribution. By the **Inverse Transformation Sampling Theorem**, we can approximate any smooth distribution.

By change-of-varibales:

$$p_x(x) = p_u(g(x))|detJ(g)(x)| = p_u(u)|detJ(f)(u)|^{-1} \tag{540}$$

where $J(f)(u) = \frac{\partial f}{\partial u}|_u$.

Taking logs of both sides:

$$logp_x(x) = logp_u(u) - log|detJ(f)(u)| \tag{541}$$

Since the base distribution is easy to evalute, all we need is that we use a flexible invertible transformation f whose Jacobian determinant $detJ(f)(u)$ can be computed efficientyl, and then one can construct complex densities $p(x)$ that allow exact smapling and efficient exact likelihood computaiton.

This is in contrast to latent variable models which require methods like variational inference to lower-bound the likelihood.

There are two ways to get these invertible mapping:

- Define a set of simple trasnformations that are invertible by design, and whose Jacobian determinant is easy to compute
- Exploit the fact that a composition of invertibel functions is invertible, and determinant are co-productive

The trasnformation f of a flow model is a **diffeomorphism**.

The density $p_x(x)$ of a flow model is also known as the **puhsforward** of the base distribution $p_u(u)$ through f, denoted:

$$p_x = f_* p_u \tag{542}$$

Training a Flow Model

There are two common applications of normalizing flows:

1. Density estimation of observed data: fitting $p_\theta(x)$ to the data and using it as an estimate of the data density
2. Variational inference: sampling from and evaluating a variational posterior $q_\theta(z|x)$ paramterized by the flow model

Constructing Flows

- Affine flows: $x = f(u) = Au + b$ for invertible square matrix A.
- Elementwise flows: $f(u) = (h(u_1), .., h(u_D))$ for a scalar-valued bijection h
- Convex combinations of strictly monotonic scalar functions
- **Coupling flow**: model dependencies between dimensions using arbitrayr non linear funcitons. Partition input into two subspaces, and assume we have a bijection on one of the coordinates. We can then define a bijection on the full space.
- **Autoregressive flows**: $x_i = h(u_i; \Theta_i(x_{1:i-1})), i = 1, ..., D$ where each output x_i depends on the corresponding input u_i and previous outputs, and Θ is an atrbirary non-linear function often paramterized by a DNN.

Any autoregressive model of continuous random variables can be reparamterized as a one-layer autoregressive flow.

Residual Flows

A **residual network** is a composition of **residual connections**:

$$f(u) = u + F(u) \tag{543}$$

where F is called the **residual block** and it comptues the difference between the output and the input, $f(u) - u$.

Theorem 0.105. *If the function F is a **contraction** (Lipschitz constant is less than 1) so that:*

$$||F(u_1) - F(u_2)|| \leq L||u_1 - u_2|| \tag{544}$$

$0 \leq L < 1$

then the residual connection is invertible.

While there is often no analytical expression for the inverse f^{-1} we can approximate using the iterative procedure:

$$u_n = x - F(u_{n-1}) \tag{545}$$

*and by **Banach Fixed Point Theorem** the sequence $u_0, u_1, ...,$ will converge to $u_* = f^{-1}(x)$ for any choice of u_0 at a rate of $O(L^n)$.*

There is no analytical expression for the Jacobian determinant (whose exact computation cost $O(D^3)$ but we have a computationally efficient sotcahstic estimator of the log Jacobian determinant as a power series:

$$log|detJ9f)| = log|det(I + J(F)| = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} tr[J(F)^k] \tag{546}$$

which converges when the matrix norm of $J(F)$ is less than 1 (guarenteed by contraction).

Energy Based Models

Sometimes its easier to specify a dsitribution in terms of constraints that valid samples must satisfy, rather than a generative process.

Energy based models can be written as a Gibbs distribution as :

$$p_\theta(x) = \frac{exp(-\mathcal{E}_\theta(x))}{Z} \tag{547}$$

where $\mathcal{E}_\theta(x) \geq 0$ is the **energy function** with parameters θ and Z is the *partition function*:

$$Z_\theta = \int exp(-\mathcal{E}_\theta(x))dx \tag{548}$$

which is constant w.r.t x.

The advantage of Energy based models over generative models is that the energy function can be any kind of function that returns a non-negative scalar - it does not need to integrate to 1.

On the other hand, computation of the likelihood and sampling from the model are intractable, and must be approximated

Maximum likelihood training

Typically, we want to optimize:

$$l(\theta) = -D_{KL}(p_{data}(x)|p_\theta(x)) + const \tag{549}$$

but this cannot be computed due to the normalizing constant.

However, the gradient of the negative log likelihood can be estimated with MCM, by decomposing:

$$\nabla_\theta logp_\theta(x) = -\nabla_\theta \mathcal{E}(x) - \nabla_\theta logZ_\theta \tag{550}$$

and the first term is tractable with auto diff. The second term can be written as the expectation:

$$\nabla_\theta logZ_\theta = \mathbb{E}_{x\sim p_\theta(x)}[-\nabla_\theta \mathcal{E}_\theta(x)] \approx -\frac{1}{S}\sum_{s=1}^{S} \nabla_\theta \mathcal{E}_\theta(x_s) \tag{551}$$

So we can reduce this to finding an efficient way to sample from the energy based model.

Score Matching

If two C^∞ function f, g have equal first derivatives everywhere then they are equal up to a constant. When f, g are log probability density function, the normalization requirement implies that they integrate to unity. Hence we learn a energy based model by approximately matching the first derivatives of its log PDF to the first derivatives of the log PDF of teh data distribution.

The **score matching** objective minimizes the **Fisher divergence** between the data distribution:

$$D_F(p_{data}(x)|p_\theta(x)) = \mathbf{E}_{p_{data}}[\frac{1}{2}||\nabla_x logp_{data}(x) - \nabla_x logp_\theta(x)||^2] \tag{552}$$

The score matching objective is only applicable under regularity condition for $logp_{data}$ which are not often satisfied (discrete data). To alleviate this, we can add noise to each datapoint.

Diffusion Models

It can be hard to convert noise into structured data, but it is easy to convert structured data into noise.

We can use a *forward process* or *diffusion* process to gradually convert the obsered data x_0 into a noisy version x_T by passing data through T steps of a stochastic encoder $q(x_t|x_{t-1})$.

After enough steps, we have $x_T \sim \mathcal{N}(0, I)$.

We then learn a *reverse process* to undo this, by passing teh noise through T steps of a decoder $p_\theta(x_{t-1}|x_t)$ until we generate x_0.

Variational Diffusion Models

The Variational diffusion model is a special case of a multi-layer VAE where each latent layer x_t has the same size as the input x_0 but has incerasing noise.

Since the encoder only adds a small amount of noise at each step, we can easily learn a decoder to invert this operation, and we don't need to worry about posterior collapse.

Encoder

$$q_{x_t|x_s)} = \mathcal{N}(x_t|\alpha_{t|s}x_s, \sigma^2_{t|s}I) \tag{553}$$

where $t = s + 1$, and $\alpha_{t|s}, \sigma_{t|s}$ are non-learned parameters.

With T layers:

$$q_{x_{1:T}|x_0} = \prod_{t=1}^{T} q(x_t|x_{t-1}) \tag{554}$$

Since the encoder is a lienar gaussian markov chain, we can compute the marginal analytically:

$$q(x_t|x_0) = \mathcal{N}(x_t|\alpha_t x_0, \sigma_t^2 I) \tag{555}$$

where

$$\alpha_{t|s} = \frac{\alpha_t}{\alpha_s} \tag{556}$$

$$\sigma^2_{t|s} = \sigma_t^2 - \alpha^{s}_{t|s}{}^2 \tag{557}$$

THe only learnable parameters for the necoder are the scalars α_t, σ_t and we set $\alpha_t = \sqrt{1 - \sigma_t^2}$ to ensure the noise is **variacne preserving**.

If we set $\alpha_t = 1$, we get **variance exploding**.

Decoder

For T layers:

$$p_\theta(x_{0:T}) = p_{x_T}[\prod_{t=0}^{T-1} p_\theta(x_t|x_{t-1})] \tag{558}$$

where $p_{x_T} = \mathcal{N}(x_T|0, I)$ and $p_\theta(x_t|x_{t+1})$ is given below.

For each step of teh decoder, we want $p_\theta(x_s|x_t)$ to be close to $q(x_s|x_t, x_0)$ since this will minimize the KL divergence in the ELBO. BY Bayes rule and markov properties:

$$q(x_s|x_t, x_0) = \frac{q(x_t|x_s)q(x_s|x_0)}{q(x_t|x_0)} \tag{559}$$

And since all terms are conditional gaussian, we can use bayes rule for gaussian systems to get exact formulas for each conditional distribution.

Model Fitting

Minimize cross entropy between empirical distirbution and model p, or equivalently maximize expected log marginal likelihood:

$$J = \int p_D(x_0) log p_\theta(x_0) dx_0 \tag{560}$$

We can get a variational lower bound on the log marginal likelihood by minimizing:

$$\mathcal{L}(x_0) = D_{KL}(q(x_T|x_0)|\pi(x_T)) + \mathbf{E}_{q(x_1|x_0)}[-log p_\theta(x_0|x_1)] + \sum_{t=2}^{T} D_{KL}(q(x_{t-1}|x_t, x_0)|p_\theta(x_{t-1}|x_t)) \tag{561}$$

the first term is the *prior loss*, the second term is the *reconstruction loss* and the final term is the *diffusion loss.*

The prior loss term is a constant, since q has no free parameters, and the diffusion loss can be optimized using the reparameterization trick, and the fact that the KL term between the gaussians is analytic in closed form.

Stochastic Estimator for Deep models

For very deep models, it is expensive to compute all T diffusion terms. We can approximate by uniformly sampling a single layer to apply the diffusion loss to.

Conditional Diffusion Models

We condition on some side information c such as a class label or text prompt.

The simplest way is to maximize the conditional likelihood $p(x|c)$, but this gives limited control over generation.

Classifier guidance leverages bayes rule to convert the conditional $p(x|c)$ into an unconditional term $p9x)$ and a likelihood $p(c|x)$. The score funciton becomes:

$$\nabla_x log p(x|c) = \nabla_x log p(x) + \nabla_x log p(c|x) \tag{562}$$

If we have a pre-trained classifier $p(c|x)$ then we can use it to convert an unconditional generative model into a conditional one.

This can have issues where the discriminative model ignore details in the input. In **classifier free guidance** we derive the classifier from the conditional mode using bayes rule to get the score of the induced classifier as :

$$\nabla_x log p(c|x) = \nabla_x log p(x|c) - \nabla_x log p(x) \tag{563}$$

This is what Open AI **DALLE-E 2** model and Google **ImageGen** model do.

Speeding up generation

Training is fast since we can just sample a random layer and optimize a squared loss between input and output. But diffusion is slow for generation , since we require T steps to get good results.

Denoising Diffusion implicit models turn off teh sampling process at all layers except the first. This adds determinism but is faster.

We can **distill** a deterministic diffusion sampler into one that takes half as many steps, and then apply recursively.

GAN

Introduction

Generative adverserial networks are **implicit generative models** which are a kind of probabilistic model without an explicit likelihood function.

They use a latent cariable z and transform it using a deterministic function G_θ. Without a likelihood, the generative procedure defines a valid density on the output space that forms an effective likeilhood:

$$z \sim q(z) \tag{564}$$

$$x = G_\theta(z') \tag{565}$$

$$q_\theta(x) = \frac{\partial}{\partial x_1} \cdots \frac{\partial}{\partial x_d} \int_{\{G_{\theta(z)} \leq x\}} q(z) dz \tag{566}$$

The final equaiton above is the definition of the transformed density $q_\theta(x)$ defined as the derivative of a CDF, and hence integrates the distributoin q over all events.

G can be specified by a deep neural network.

This is called **likelihood-free inference**, which forms the basis of **Approximate Bayesian Computation.**

Learning by comaprison

We can look at ratios (class probability estimation or f-divergences) or differences (integral probability metrics or moment matching) on samples purely generated from the model.

We want objectives $D(p^*, q)$ s.t.:

1. Provide guarantees about learning the data: $argmin_q D(p^*, q) = p^*$
2. Can be evaluated only using samples from the data and model distribution
3. Are cheap to evaluate

Many measures (KL cannot be evaluted only using samples) or wasserstein (expensive) fails.

The main approach is *to approximate the desired quantity through optimization by introducing a comparison model* often called a **discriminator or critic** D such that:

$$D(p^*, q) = argmax_D \mathcal{F}(D, p^*, q) \tag{567}$$

where $\mathcal{F}$ is a functional that depends on p^*, q only through samples.

The model and critic are paramterized with θ, ϕ repsectively, the critic to optimize an expectation of the above:

$$\mathcal{F}(D_\phi, p^*, q_\theta) = \mathbf{E}_{p^*(x)} f(x, \phi) + \mathbf{E}_{q(z)} g(G_\theta(z), \phi) \tag{568}$$

by monte carlo, and the model, the exact objective $D(p^*, q_\theta$ is replaced with the tractable approximation provided through the use of the discriminator D_ϕ.

We combin and use any proper scoring rule. For example, with binary cross entropy:

$$V(q_\theta, p^*) = argmax_\phi \frac{1}{2} \mathbf{E}_{p^*(x)} log D_\phi(x) + \frac{1}{2} \mathbf{E}_{q_\phi(x)} log(1 - D_\phi(x)) \tag{569}$$

This results in a **minmax optimization**:

$$min_\theta max_\phi \frac{1}{2} \mathbf{E}_{p^*(x)}[log D_\phi(x)] + \frac{1}{2} \mathbf{E}_{q(z)}[log(1 - D_\phi(G_\theta(z)))] \tag{570}$$

By the theory of **convex conjugate functions**, we can get a variational lower bound on any f-divergence, which leads to **f-GANS**.

As we know, their is an isomorphism between the set of scoring rules and f-divergences so they are mathematically equivalent.

Alternatively, we can compare densities using their differences instead of ratios, leading to *integral probability metrics*

$$I_{\mathcal{F}}(p^*(x), q_\theta(x)) = sup_{f \in \mathcal{F}} |\mathbf{E}_{p^*(x)} f(x) - \mathbf{E}_{q_\theta(x)} f(x)| \tag{571}$$

This is nice because divergence require overlapping support. Hence *f-divergences cannot distinguish between different model distributions when they do not have overlapping support with the data distribution.*

The functions f is a test function that will take thew role of the critic. To do this, we must define the class of real valued measurable function $\mathcal{F}$ over which the supremum is to be taken.

A popular choose is the set of 1-Lipschitz function which leads to the **wasserstein distance**, and hence **Wasserstein GAN**. We can keep the critic 1-lipschitz by applying gradient penalties or spectral normalization methods.

Alos, we can choose any class of functions from a reproducing kernel hilbert space, leading to the **maximal mean discrepancy formulation** where the critic is parameterized by a kernel function.

Model

Instead of having the generator minimize the probability that the discriminator classifies its samples as fake, we can maximize the probabilitity that the discriminator calssifies its samples as real. This is a *non-saturating loss* which have a strongler learning signal and can help early in training.

We try to optimize the discriminator at each step of optimziation of the generator (K steps), and each set of parameters is frozen while the other optimizes.

The disriminator can just use monte carlo, but the generator needs to use the reparameterization trick to differentiate through the noise.

Gans are known to suffer from **mode collapse** where the generator converges to a distribution which does not cover all the modes of the data, thus underfiting.

If the discriminator is optimal, the the generator is optimization the divergence between the data and model distribution, but in practice convergence is non-stable, and the nash-equilibrium has guarentees that are not realized in practice.

Conditional GAN

We want to generate conditional distributions of the form $p^*(x|y)$ and for this we need **paired data**.

It is important for the critic to provide conditioning information.

Neural architectures in GAN

Critic architectures must be selected with the appropriate inductive biases in mind since they are a parameterization of the loss function that the generator is trying to optimize.

- Batch normalization for both actor critic
- Conv network
- Replace pooling layers with strided convolutions

- Using relu in the generator and leaky relu in the discriminator
- Residual convolution layers
- Attention
- Regularization by adding noise to discriminator input, adding dropout and penalizing the norm of the discriminator gradient
- Huge batch sizes

Discovery

Introduction

The goal is to *discover something about the dataset.*

A common approach is to use a **latent variable model** in which we maek the assumption that the obsereved data x was generated by some underlying **latent factors** z.

Latent Factor Models

A **latent variable model** is any probabilistic model in which some variables are latent. For example a mixture model $p(x) = \sum_k p(x|z=k)p(z=k)$ where z is an indicator variable that specifeis which mixture component to use for generating x.

The general use case is:

$$z \sim p(z) \tag{572}$$
$$x|z \sim ExpFam(x|f(z)) \tag{573}$$

Where f is the **decoder** and $p(z)$ is the prior.

Mixture models

Convex combination:

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k p_k(x) \tag{574}$$

Which can be seen as a **hierarhical latent variable model** with $z \in \{1, ..., K\}$ a categorical random variable.

Hence:

$$p(z|\theta) = Cat(z|\pi) \tag{575}$$
$$p(x|z=k,\theta) = p(x|\theta_k) \tag{576}$$

A **gaussian mixture model** is defined as :

$$p(x) = \sum_{k=1}^{K} \mathcal{N}(x|\mu_k, \Sigma_k) \tag{577}$$

Theorem 0.106. *For enough mixture components, a GMM can approxiamte any smooth distribution on* $\mathbb{R}^d$

GMM are often used for **unsupervised clustering** of real valued data $x_n \in \mathbf{R}^D$.

- First we fit the model by computing the MLE: $\hat{\theta} = argmaxlogp(\mathcal{D}|\theta)$
- Then we associate each data point x_n with a discrete factor by looking at the posterior:

$$r_{nk} = p(z_n = k|x_n, \theta) = \frac{p(z_n=k|\theta)p(x_n|z_n=k,\theta)}{\sum_{k'=1}^{K} p(z_n=k'|\theta)p(x_n|z_n=k',\theta)} \tag{578}$$

  where r_{nk} is the **responsibility of cluster k for data point n**

- Given responsibilities we can compute cluster assignment

$$\hat{z}_n = argmax_k r_{nk} \tag{579}$$

- *If we have a uniform prior on z*, and we use spehrical gaussians, the hard clsutering reducted to **K-means** where:

$$z_n = argmin_k ||x_n - \hat{\mu}_k||_2^2 \tag{580}$$

Using mixture models for classification

We can model the class conditional desnity $p(x|y=c)$ and derive the posterior using bayes rule:

$$p(y=c|x) = \frac{p(y=c)p(x|y=c)}{Z} \tag{581}$$

where $p(y=c) = \pi_c$ is the prior on the class label.

Using a generative model to perform classification can be useful when we have missing data, since we can compute $p(x^v|y=c) = \sum_{x^m} p(x^m, x^v|y=c)$ to compute the marginal likelihood of the visible features x^v.

It is also useful for semi-supervised learning for the same reason.

Numerical issues: to compute the class conditional log likelihood $l_c = logp(x|y-c)$ we can use the **log-sum-exp trick to aovid numerical underflow**.

Factor Analysis

We consider a simple latent factor model in which the prior $p(z)$ is gaussian, and the likelihood $p(x|z)$ is also gaussian using a lienar decoder for the mean. This family includes special cases such as PCA.

$$p(z) = \mathcal{N}(z|\mu_0, \Sigma_0) \tag{582}$$

$$p(x|z,\theta) = \mathcal{N}(x|Wz+\mu, \Psi) \tag{583}$$

where W is a $D \times L$ matrix known as the **factor loading matrix** and Ψ is a diagonal $D \times D$ covariance matrix.

Factor analysis can be though of as a low-rank version of a gaussian distribution. In general, FA approximates the covariance matrix of the joint gaussian using the low rank decomposition:

$$C = Cov[x] = WW^T + \Psi \tag{584}$$

which has $O(LD)$ parameters.

Posterior

WE can compute the posterior over the latent codes $p(z|x)$ using Bayes rule for gaussians, and avoid inverting the covariacne matrix C by using the **matrix inversion lemma**.

Compute the likelihood

We want to compute the log marginal likelihood given by:

$$logp(x|\mu, C) = -\frac{1}{2}[Dlog(2\pi) + logdet(C) + (x-\mu)^T C^{-1}(x-\mu)] \tag{585}$$

where $C = WW^T + \Psi$

We can fit using **Expectation Maximization**.

The paramters of FA model are unidentifiable since the they are isomoprhically equivalent under orthogonal rotations of W. This is called the **factor rotationsp roblem**.

To break symmetry, we can *force W to have orthogonal columns* as in **PCA**.

Probabilistic PCA

Consider W orthogonal, $\Psi = \sigma^2 I$ so $p(x) = \mathcal{N}(x|\mu, C)$ for $C = WW^T + \sigma^2 I$. This is called **probabilistic principal component analysis**.

- Advantag eover FA is that MLE has closed form
- Advantage over non-probabilistic PCA is that the model deines a proper likelihood, which makes it easier to extend, for example, by creating mixtures.

Note: PCA is recovered in the noise-free limit where $\sigma^2 = 0$.

In this case, the MLE covariance is a rank L approximation.

To use PPCA we need to compute the posterior mean $\mathbf{E}[z|x]$ which is the equivalent of the pCA ENCODER.

Again, in the noise free limit, we are left with the orthogonal projection of the data into the latent space:

$$\mathbf{E}[z|x] = (W^T W)^{-1} W^T (x - \mu(x)) \tag{586}$$

Mixture of Factor Analysers

The factor analysis model assuems the osberved data can be modeled as arising from a linear mapping form a low-dimensionals et of Gaussian factors. WE can relax this by just assuming it is *locally linear*, so the overal model is a weighted combination of FA Models, called a **mixture of factors**.

The overal model for the data is a mixture of lienar manifolds, each of which approximate a curved manifold.

Factor analysis with exponential family likelihoods

To model non-real valued data, such as binary or categorical we can simply replace the gaussian output distribution with a suitable exponential family distribution.

Representation Learning

Interpretability

Action

Decision Making Under Uncertainty

Bayesian Decision Theory

In decision theory, we assume the **agent** has a set of possible actions $\mathcal{A}$, to gain an objective, which will depend on the underlying **state** $h \in \mathcal{H}$. We can encode this into a **loss function** $l(h, a)$.

Then we can computed the **posterior expected loss or risk** for each:

$$R(a|x) = \mathbf{E}_{p(h|x)}[l(h, a)] = \sum_{h \in \mathcal{H}} l(h, a)p(h|x) \tag{587}$$

The **bayes optimal policy** minimizes the above:

$$\pi^*(x) = argmin_{a \in \mathcal{A}} \mathbb{E}_{p(h|x)}[l(h, a)] \tag{588}$$

and maximizes the *expected utility*.

Contextual Bandit

In a **sequential decision making problem**, we have to optimize over a time horizon. The simples kind is the **bandit problem**.

In a **multi-armed bandit** the agent can choose an **action** from a **policy** $a_t \sim \pi_t$ at each step, after which it receives a **reward** sampled from the **environment** $r_t \sim p(a_t)$.

We can extent to a **contextual bandit** in which the input to the policy at each step is a context $s_t \in \mathcal{S}$ that evolves according to some process $s_t \sim p(s_t|_{s_{1:t-1}})$.

In the **finite horizon case** the goal is to maximize the cumulative reward:

$$J = \sum_{t=1}^{T} \mathbf{E}[r_t] \tag{589}$$

In the infinite horizong case, we need to bound the objective, so we introduce a **discount factor** $0 \leq \gamma \leq 1$ and then optimize:

$$J = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{E}[r_t] \tag{590}$$

Exploration Exploitation Tradeoff

The agent needs to try multiple state action combinations in order to collect enough data so it can reliable learn the reward function; it can then exploit its knowledge by picking the predicted best action. If the agent exploites too quick, it is exploiting a model that is not optimal.

This is differnet from supervised learning: here, the *the model creates the data*.

The optimal solution: let us denote the posterior over the parameters of teh reward by $b_t = p(\theta|h_t)$, i.e. the ebleif states, where h_t is the history of observations. By bayes rule, we can update:

$$b_t = Bayes(b_{t-1}, a_t, r_t) \tag{591}$$

Upper confidence bounds (UCB)

The optimal solution to explore-exploit is intractable. The UCB method involves **optimism in the face of uncertainty**. We select the aciton greedily, but based on optimisitc estimates of tehir reward.

To use this method, the agent maintains an optimisitic reward estimate $\hat{R}_t$ so that $\hat{R}_t(s_t, a) \geq R(s_t, a)$ and then chooses greedily:

$$a_t = argmax_a \hat{R}_t(s_t, a) \tag{592}$$

To obtain the optimistic reward function we can:

- Frequentist approach:via the **concentration inequality**
- Bayesian inference

Thompson sampling

A common alternative to UCB is **Thompson sampling** or **probabilitity matching**. In this approach we define the policy at step t to be $\pi_t(a|s_t, h_t) = p_a$:

$$p_a = \int \mathbb{I}(a = argmax_{a'} R(s_t, a'; \theta) p(\theta|h_t) d\theta \tag{593}$$

If the posterior is uncertain, the agent will sample many different actions, automatically resulting in exploration. As the ucnertaintty decreases, it will start to exploit its knowledge.

To implement, use a single MOnte carlo sample $\hat{\theta}_t \sim p(\theta|h_t)$, and plug this into our reward model:

$$a_t = argmax_{a'} R(s_t, a'; \hat{theta}_t) \tag{594}$$

This method achieves optimal (logarithmic) regret.

The **regret** is defined as the difference between the expected reward under the agent policy and the oracle policy π^* which knows the true reward function. We often care about the cumulative regret. Under typical assumption that reward are bounded L_T is at most linear in T. If the agent policy converges to the optimal , then regret is sublinear.

Both UCB and Thompson sampling achieve sub linear regret, and both are optimal in the sense that their regrets are non-improvable, matching lower bounds.

MDP

A **markov decision process** is a tuple $\langle \int, \mathcal{A}, p_T, p_R, p_0 \rangle$ where S are states, A is reward, p_T is the transition modelm, p_R is the reward model, and p_0 is the initial state distribution.

The reward function is given by:

$$R(s, a) = \mathbb{E}_{p_T(s'|s,a)}[\mathbb{E}_{p(r|s,a,s')}[r]] \tag{595}$$

We want the optimize the discounted cumualtive reward.

A **partially observable MDP** generalizes by relaxing the assumption that the agent sees the hidden world state s_t directly, and instead sees a noisy observation generated from a hidden state:

$$x_t \sim p(\cdot|s_t, a_t) \tag{596}$$

POMDP are much harder to solve.

Episodes and Returns

An MDP defines how a trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$ is stochastically generated. An **episode** is a single interaction with the process until we reach a **terminal state**.

Let τ be a trajectory of length T, we define the **return** for state at time t to be the sum of expected rewards obtained going forward, multiplied by a discount $\gamma \in [0, 1]$

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...\gamma^{T-t-1} r_{T-1} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{597}$$

Also called the **reward to go**.

The return satisfies the recursive relationship:

$$G_t = r_t + \gamma G_{t+1} \tag{598}$$

Value functions

Let π be a policy. The **value function** is:

$$V_\pi(s) = \mathbf{E}_\pi[G_0|s_0 = s] = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t|s_0 = s] \tag{599}$$

the expected return obtained if we state in state s and follow π.

The **action value or Q function** is:

$$Q_\pi(s, a) = \mathbf{E}_\pi[G_0|s_0 = s, a_0 = a] = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t|s_0 = s, a_0 = a] \tag{600}$$

which represents the expected return if we start by taking action a in state s and then follow π.

The **advantage function** is:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{601}$$

which is the beenfit of picking action a relative to baseline return of following π.

Note that the expected advantage is 0 since:

$$V_\pi(s) = \mathbb{E}_{\pi(a|s)}[Q_\pi(s, a)] \tag{602}$$

Optimal value functions and policies

Suppose π_* is a policy such that $V_{\pi_*} \geq V_\pi$ for all $s \in \mathcal{S}$, then it is an **optimal policy**.

We call $V_* = v_{\pi_*}$ the **optimal value function** and Q_{π_*} the **optimal action value function**.

Theorem 0.107. *Any finite state MDP must have at least one determinsitic optimal policy*

Theorem 0.108 (Bellman)**.**

$$V_*(s) = max_a R(s, a) + \gamma \mathbb{E}_{p_t(s'|s,a)}[V_*(s')] \tag{603}$$
$$Q_*(s, a) = R(s, a) + \gamma max_{a'} \mathbb{E}_{P_t(s'|s,a)}[Q_*(s', a')] \tag{604}$$

Conversely, the optimal value functions are the only solutions to these equations.

The discrepancy between the equality above are called the **Bellman residual**.

Given the optimal value function, we can derive an optimal policy:

$$\pi_*(s) = argmax_a Q_*(s,a) = R(s,a) + \gamma max_{a'} \mathbb{E}_{P_t(s'|s,a)}[V_*(s')] \tag{605}$$

Solving for these quantities is called **policy optimization**.

In contrast, solving for V_π, Q_π for a given policy π is called **policy evaluation**.

Planning in an MDP

The problem of **planning** is finding an optimal policy when the MDP is known, and is based on **dynamic programming** and **linear programming**.

Value iteration starts from an initial value function V_0 an iteratively updates:

$$V_{k+1}(s) = max_a[R(s,a) + \gamma \sum_{s'} p(s'|s,a)V_k(s')] \tag{606}$$

which is called the **bellman backup update rule**, which is exactly the RHS of teh bellman optimality equation.

Fundamentally, this is a contraction:

$$max_s|V_{k+1}(s) - V_*(s)| \leq \gamma max_s|V_k(s) - V_8(s)| \tag{607}$$

so every iteration will reduce the maximum value function error by a constant factor, and then we can extract policy.

Another effective DP method for computing optimal policy given MDP is called **policy iteration**. Each iterative step performs **policy evaluation** and **policy improvement**.

- Policy Evaluation: compute the value funciton by solving a linear system
- Policy Iteration: given the value function of the policy, derive a better policy by computing a deterministic policy π' that acts greedily with respect to our current value function.

Iterative apply the chain of updates until the policy is greedy with respect to its own value function. In this case, it is optimal.

Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies, and every iteration stricly improves, we have finite convergence.

Reinforcement Learning

Introduction

Reinforcement learning is a paradigm of learning where an agent sequentially interacts with an initially unknonwn environment. The interaction is called a **trajectory**.

Let $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ...s_T$ be a trajectory of length T, consisting of a sequence of states s_t, actions a_t and rewards r_t. The goal of teh agent is to optimize the action selection policy, so tha thte discounted cumulative reward:

$$G_0 = \sum_{t=0}^{T-1} \gamma^t r_t \tag{608}$$

is maximized for some given **discount factor** $\gamma \in [0,1]$.

In general G_0 is a random variable. We typically maximize its expectation, but other methods can be useful in the presence of risk.

We focus on MDP, where generative model for the trajectory can be factored into single step models. When the parameters are known, solving for an optimal policy is called **planning**, otherwise RL may be used to obtain an optimal policy from trajectories.

In **model free** RL, we try to learn the policy without explicitly representing and learning the models. In **model based RL** we first solve for the model, and then use a planning algorithm on the learned model to solve the policy.

Value based methods try to learn the optimal Q function from experience, and then derive a policy from it. Typically, a function approximator is used Q_w to represent the Q function.

Given a transition (s, a, r, s') we define the **temporal difference** or **TD Error** as:

$$r + \gamma max_{a'} Q_w(s', a') - Q_w(s, a) \tag{609}$$

The expected TD error is teh bellman error evaluated at (s, a). Thus,$Q_w = Q_* \implies TDError = 0$ in expectation. Else the error in the signal for teh agent to change w to reduce.

This kind of update is known as **bootstrapping**.

In **policy search** we try to directly maximize J_{π_θ} wrt the policy parameters θ. If J_{π_θ} is differntiable wrt θ, we can use stochastic gradient ascent to optimize, which is known as **policy gradient**. The basic idea is to perform **monte carlo rollouts** in which we sample trajectories by interacting with the environment, and then use the score function estimator to estimate $\nabla_\theta J(\pi_\theta)$. These methods have provably convergence to local optima, and work out of the box for continuous action spaces, however the score function estimator has high variance, so the convergence is slow.

One way to reduce the vcariance is to learn an approximate value function $V_w(s)$ and then use it as a baslien in the score funciotn estimator. This leads to **actor critic methods**.

Value based methods and policy search methods can be **very sample inefficient** meaning they may need to interact with the environment for a long time before finding a good policy. If an agent has a prior knowledge of the MDP, it can be more sample efficient to learn the model, and the compute an optimal policy wrt to our modelling estimates. This is called **model-based RL**. This requires a mdp model of $p_T(s'|s, a)$ and $R(s, a)$ using DNN, given a collection of transitions, using supervised learning methods.

A common heuristic is $\varepsilon - greedy$ in which we pick the greedy action wrt the current model $a_t = argmax_a \hat{R}_t(s_t, a)$ with probability $1 - \varepsilon$ and a random action with probability ε. While this ensures eventual exploration, it is suboptimal since it samples every aciton with at least constant probability $\frac{\varepsilon}{|A|}$ which is bad for high dimensional spaces.

The **boltzmann policy** can be more efficient by assigning higher probabilities to epxlore more promising actions:

$$\pi_\tau(a|s) = \frac{exp(\hat{R}_t(s_t, a))/\tau}{\sum_{a'} exp(\hat{R}_t(s_t, a'))/\tau} \tag{610}$$

where $\tau > 0$ is a temperature parameter that controls how entropic the distributing is (closer to zero implies greedy, higher implies more uniform, and hence more exploration).

UCB and Thompson sampling extends to MDP but we need to also account for uncertainty in transitions (not just reward). This typically involves a **count based** exploration scheme or exploraiton bonus such as:

$$\hat{r} = r + \alpha/\sqrt{N_{s,a}} \tag{611}$$

Value Based RL

Monte Carlo RL

Recall $Q_\pi(s, a) = \mathbb{E}[G_t|s_t = s, a_t = a]$. A simple way to estimate this is to take aciton a, and then sample the rest of the trajectory according to π, and then compute the average sum of discounted reward. This is called **monte carlo estimation of teh value function**.

We can use this together with policy iteration to learn an optimal policy, called **Monte Carlo Control**:

- Compute a new improved policy using $\pi_{k+1}(s) = argmax_a Q_k(s, a)$
- Estimate Q_{k+1} with MC estimation

To ensure convergence, we need to explore, which we can do with an $\varepsilon - greedy$ policy, with gradually decrease in ε.

TD Learning

Monte Carlo control results in an estimator $Q_\pi(s, a)$ with very high variance, since it has to unroll many trajectories, whose returns are a sum of many random rewards generated by stochastic state transitions.

A more efficient technique is **TD learning**. The idea is to incrementally reduce the Bellman error for sampeld states based on transitions instead of a long trajectory. Given (s, a, r, s') for $a \sim \pi(s)$ we change the estimate $V(s)$ so that it moves toward the bootstrapping target:

$$V(s_t) = V(s_t) + \eta[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{612}$$

where η is the learning rate. With parametric value functions:

$$w = w + \eta[r_t + \gamma V_w(s_{t+1}) - V_w(s_t)]\nabla_w V_w(s_t) \tag{613}$$

Despite how this looks, this is *not* SGD on any objective function, since we use **bootstrapping** in which the estimate $V_w(s_t)$ is updated to approach a target $r_t + \gamma V_w(s_{t+1})$ which is defined by the value function estimate itself.

TD Learning with Eligibility Traces

A key difference betwen TD and MC is how we estimate returns. Given a trajectory $\tau = (s_0, a_0, r_0, s_1, ..., s_T)$ TD estimates the return form state s_t by a one step look ahead:

$$G_{t:t+1} = r_t + \gamma V(s_{t+1}) \tag{614}$$

In contrast, MC waits until the end of the episode, then uses:

$$G_{t:T} = r_t + \gamma r_{t+1} + ... + \gamma^{T-t-1} r_{T-1} \tag{615}$$

It is possible to interpolate between these by performing an n-step rollout, and then using the value function to approxiamte the resturn for the rest of the trajectyory, giving the n-step estimate:

$$G_{t:t+n} = r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \tag{616}$$

The corresponding n-step TD updateis:

$$V(s_t) = V(s_t) + \eta[G_{t:t+n} - V(s_t)] \tag{617}$$

Rather than picking a particualar n step lookahead value, we can take a convex weighted average of all possible values, with a single parameter $\lambda \in [0, 1]$ s.t:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} {}^{n-1} G_{t:t+n} \tag{618}$$

which is called **λ-return**.

The geometric weighting can be efficiently implemented using **eligibility traces**. This is called **TD(λ)** and can be combined with many other methods.

SARSA: on policy TD control

TD learning is for policy evalution, as it estimates the value function for a fixed policy. In order to find the optimal policy, we may use the algorithm as a building block inside generalized policy iteration.

The agent follows π in every step to choose actions, and upon a transition (s, a, r, s') the TD update:

$$Q(s,a) = Q(s,a) + \eta[r + \gamma Q(s', a') - Q(s,a)] \tag{619}$$

where $a' \sim \pi(s')$ is the aciton the agent will take. After Q is updated for policuy evaluation, π is chosen to be greedy wrt the new Q (policy improvement).

This is called **SARSA**. If we are greedy in the limit with infinite exploration **GLIE** (for example ε-greedy with decaying ε) this will convergence.

Q-learning: off policy TD control

SARSA is an **on-policy** algorithm, which means it learns the Q function for the policy it is currently using, which is typically not the optimal policy. With a simple modification, we can convert it to an **off-policy** algorithm that learns Q_* even if a suboptimal policy is used to choose actions.

We replace the sampled next action $a' \sim \pi(s')$ with a greedy action in s':

$$a' = argmax_b Q(s', b) \tag{620}$$

leading to :

$$Q(s,a) = Q(s,a) + \eta[r + \gamma max_b Q(s', b) - Q(s,a)] \tag{621}$$

which is called **Q learning**.

Since it is **off policy**, the method can use (s, a, r, s') triples from any data source, such as older versions of the policy, or even log data from an existing non RL system.

Double Q learning

Standard Q learning has the **maximization bias**:

$$\mathbf{E}[max_a X_a] \geq max_a \mathbb{E}[X_a] \tag{622}$$

and so if we pick actions greedily according to their random scores, we might pick a wrong action jsut because random noise.

One solution is to use two seperate Q functions Q_1, Q_2 one for selecting the greedy action, and hte other for estimating the corresponding Q value, so for a transition (s, a, r, s') we put:

$$Q_1(s,a) = Q_1(s,a) + \eta[r + \gamma Q_2(s', argmax_{a'} Q_1(s', a')) - Q_1(s,a)] \tag{623}$$

and the swap the roles of Q_1, Q_2.

This is **double q learning**.

Deep Q network DQN

When function approxiamtion is sued for Q learning, we call it **DQN**. To help with stability, there are multiple techniques;

- Use an **experience replay buffer** which stores recent transition tuples (s, a, r, s')
- Sample random transition from this pool to improve data efficiency (we can use the same transition multiple times), an dimprove stability (by reducing correlation of data samples)
- Regress the Q network to a **frozen target network** computed form an earlier iteration, rather than trying to chase a constantly moving target.

- We maintain an extra frozen copy fo the Q netowork Q_{w-} and bootstrap is for targets for training Q_w which is called **fitted value iteration**
- We can use a **prioritized experience replay** which samples in priority of transitions with larger TD error under the current Q function
- Learn a value function V_w and an advantage function A_w with shared parameters, instead of learning Q. This is called **dueling DQN** and is more sample efficient

Rainbow method combines all three methods, along with multi-step returns, distribution RL (predicts distribution of returns not just expected return) and noisy nets (adds random noise to network weights to encourage exploration. This is a SOTA value based method.

Policy Based RL

Value based methods have three main disadvantages

- Don't work well in continuous action spaces
- May diverge for function approximation
- TD style updates are not directly related to expected return

Policy search methods instead directly optimize the parameters of the policy as to maximize expected reutnr.

Policy gradient theorem

The objective is the expected return of a policy:

$$J(\pi) = \mathbb{E}_{p_0,\pi}[G_0] = \mathbb{E}_{p_0(s_0)}[V_\pi(s_0)] = \mathbb{E}_{p_0(s_0)\pi(a_0|s_0)}[Q_\pi(s_0,a_0)] \tag{624}$$

We consider parametric policies π_θ. The gradient of the objective wrt to θ is :

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{p_0(s_0)}[\sum a_0 \nabla \pi_\theta(a_0|s_0) Q_{\pi_\theta}(s_0,a_0)] + \mathbb{E}_{p_0(s_0)\pi_\theta(a_0|s_0)}[\nabla_\theta Q_{\pi_\theta}(s_0,a_0)] \tag{625}$$

Recursively expanding $\nabla_\theta Q_{\pi_\theta}(s_0,a_0)$ using the bellman equations yields:

$$\nabla_\theta J(\pi_\theta) = \frac{1}{1-\gamma}\mathbb{E}_{p^\infty_{\pi_\theta}\pi_\theta(a|s)}[\nabla_\theta log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)] \tag{626}$$

where $p^\infty_{\pi_\theta} = (1-\gamma)\sum_{t=0}^\infty \gamma^t p_t(s)$ is the normalized discounted state visitation distirbution.

This is the **policy gradient theorem**.

In practice, estimating the policy gradient using the above has high variance. A baseline $b(s)$ can be used for variance reduction:

$$\nabla_\theta J(\pi_\theta) = \frac{1}{1-\gamma}\mathbb{E}_{p^\infty_{\pi_\theta}\pi_\theta(a|s)}[\nabla_\theta log \pi_\theta(a|s)(Q_{\pi_\theta}(s,a) - b(s))] \tag{627}$$

A common choice of baseline is $b(s) = V_{\pi_\theta(s)}$.

Reinforce

One way to apply the policy gradient theorem to optimize a policy is to use stochastic gradient ascent. Suppose $\tau = (s_0, a_0, r_0, s_1, .., .s_T)$ is a trajecotry with $s_0 \sim p_0$. Then:

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta log \pi_\theta(a_t|s_t) \tag{628}$$

where G_t is teh return.

With a baseline, we get the **REINFORCE** algorithm:

$$\theta = \theta - \eta \sum_{t=0}^{T-1} \gamma^t (G_t - b(s_t)) \nabla_\theta log \pi_\theta(a_t|s_t) \tag{629}$$

Intuitivaly, we compute the sum of discounted future rewards induced by a trajectory, compared to a baseline, if this is positive, we increase θ so as to make this trajectory more likely, and reduce otherwise.

Actor Critic

An **actor critic method** uses the policy gradient method, but where the expected return is estimated using TD elarning instead of MC rollouts. The *actor* refers to the policy and the *critic* refers to teh value function.

The use of bootstrapping in TD updates allows for more efficient learning of the value function. Also, it allows us to learn fully online, without the need to wait until the end of the trajectory before updating the parameters.

For example, if we use the one step $TD(0)$ method to estimat the return, we replace:

$$G_t = G_{t:t+1} = r_t + \gamma V_w(s_{t+1}) \tag{630}$$

If we further use $V_w(s_t)$ as the baseline, we are in effect looking at:

$$r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) \tag{631}$$

which is a single sample approximation to the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. This method is called **Advantage Actor Critic** or **A2C**.

If we run the actors in parallel and asynchronously update their shared parameters, the method is called **Asynchronous Advantage Actor Critic** or **A3C**.

More generally, we can use the n-step rollout, $G_{t:t+n} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-1} r_{t+n+1} + \gamma^n V_w(s_{t+n})$ and obtain an n-step advantage estimate:

$$A^n_{\pi_\theta}(s_t, a_t) = G_{t:t+n} - V_w(s_t) \tag{632}$$

The n steps of actual rewards are unbiased, but have high variance. Bu contrast, $V_w(s_{t+n+1})$ has lower variance but is biased. By changing n, we control the **bias variance tradeoff**.

If we take the weighted advantage as in $TD(\lambda)$ we get:

$$A^\lambda_{\pi_\theta}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{633}$$

$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t) \tag{634}$$

where $\lambda \in [0, 1]$ controls the bias variance tradeoff (larger values decrease bias but increase variance.

We can implement **actor critic with generalized advantage estimation GAE** but combining with policy gradient theorem.

Bound optimization methods

Policy gradient methods do not having a monotocally increasing objective and can collapse if the learning rate is not small enough. To guarentee montonic improvement, we require that our policy update is conservative, by ensuring the divergence between policies is not too large. This is the idea of **Trust region policy optimization**, which uses the KL divergence as a measure:

$$J^{TRPO}(\pi') = J(\pi) + \frac{1}{1-\gamma} \mathbb{E}_{p^\infty_\pi(s)\pi(a|s)}[\frac{\pi'(a|s)}{\pi(a|s)} A_\pi(s, a)] - \frac{\varepsilon\gamma}{(1-\gamma)^2} max_s D_{KL}(\pi(s)|\pi'(s)) \tag{635}$$

where $\varepsilon = max_{s,a}|A_\pi(s,a)|$

This may be overly conservative, so in practice we can approximate:

- Replace point-wise maximum KL divergence by an average KL
- Maximize wrt to a policy π' lying in a KL ball centered at π

$$argmax_{\pi'} L(\pi') st \mathbb{E}_{p_\pi^\infty(s)}[D_{KL}(\pi(s)|\pi'(s)] \leq \delta \tag{636}$$

The trust region method, using a KL penalty at each step is equivalent to natural gradient descent. This is important because a step of size η in parameter space does not always correspond to a step of size η in policy space.

Another approach is **Proximal Policy Optimization** which uses a clipped objective:

$$J^{PPO}(\pi') = \frac{1}{1-\gamma}\mathbb{E}_{p_\pi^\infty(s)\pi(a|s)}[ccc(\frac{\pi'(a|s)}{\pi(a|s)})A_\pi(s,a)] \tag{637}$$

where

$$ccc(x) = clip(x, 1-\varepsilon, 1+\varepsilon) \tag{638}$$

so as to ensure:

$$|ccc(x) - 1| \leq \varepsilon \tag{639}$$

Determinisitc policy gradient methods

Assume now we have a deterministic policy that predicts a unique action for each state $a_t = \mu_\theta(s_t)$ instead of $a_t \sim \pi_\theta(s_t)$. The objective is:

$$J(\mu_\theta) = \frac{1}{1-\gamma}\mathbb{E}_{p_{\mu_\theta}^\infty(s)}[R(s, \mu_\theta(s))] \tag{640}$$

The **deterministic policy gradient theorem** yields:

$$\nabla_\theta J(\mu_\theta) = \frac{1}{1-\gamma}\mathbb{E}_{p_{\mu_\theta}^\infty(s)}[\nabla_\theta \mu_\theta(s) \nabla_a Q_{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}] \tag{641}$$

DDPG uses DQN along with the deterministic policy gradient theorem.

Model Based RL

Model Predictive Control

We can while in state s_t use a model to predict future states and rewards that might follow for each possible sequence of future actions we might pursue. We take the action that is most promising, and repeat:

$$a_{t:t+H-1}^* = argmax_{a_t:t+H-1}\mathbb{E}[\sum_{h=0}^{H-1} R(s_{t+h}, a_{t+h}) + \hat{V}(s_{t+H})] \tag{642}$$

Here H is called the **planning horizon** and $\hat{V}_{s_{t+H}}$ is an estimate of the reward-to-go at the end of the H-step look ahead.

This is known as **model predictive control**.

Monte Carlo tree search MCTS learns a value function for each encountered state, rather than relying on manually designed heuristics, which is the basis for **Muzero** and **Stochastic Muzero** wher ethe world

model is learned. The action-value functions for the intermediate nodes in the search tree are represented by deep neural networks, and updated using TD methods.

For continuous acitons, we cannot enumerate all possible branches in the search tree, so we must view the above as a non-linear program. Instead we ahve to resort to a **linear gaussian controller**.

MBRL using Latent Variable Models

Learn latent variable models rather than trying to predict dynamics directly in the observed space.

World models learn a generative model such that the model can be used to train a policy entirely in simulation.

Off policy learning

Offline reinforcement learning ate much more data efficient than their on policy counterparts. A key challenge is that the data distribution is typically different form the desired one. If we are to estimate expected returns, but the trajectories are generated by a different policy, their is bias between the expectation under the current policy, and the one that generated the data.

The off-policy data is assumed to be a collection of trajectories $\mathcal{D} = \{\tau^i\}$ where each trajectory is a sequence: $\tau^i = (s_0^i, a_0^i, r_0^i, ...)$ where the rewards and next states are sampled acording to the reward, transition models, and some **behaviour policy** π_b, which is different from the **target policy** π_e that the agent is evaluating or improving.

We can use importance sampling to correct for distirbutional mismatches in off policy data.

For example, suppose we want to estimate the target policy value $J(\pi_e)$ with fixed horizon T. Correspondingly the trajectories in $\mathcal{D}$ are also of length T. Then the **IS Off policy estimator** is given by:

$$\hat{J}_{IS}(\pi_e) = \frac{1}{n}\sum_{i=1}^{n} \frac{p(\tau^i|\pi_e)}{p(\tau^i|\pi_b)} \sum_{t=0}^{T-1} \gamma^t r_t^i \tag{643}$$

which is unbiased. The **importance ratio** is used to compensate for the fact that the data is sampled from π_b not π_e. Furthermore, the ratio *does not* depend on the MDP models.

This makes it easy to use if the target policy is known.

In practice it is unstable due to high variance. One improvement is to use the **per-step importance sampling** variant, that has lower variance:

$$\hat{J}_{PDIS}(\pi_e) = \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\prod_{t'\leq t}\rho_t'(\tau^i)\gamma^t r_t^i \tag{644}$$

The IS and DR approaches suffer from the **curse of horizon**, *the variance grows exponentially in the trajectory length.*

Control as inference

Control as inference reduces policy optimization to probabilistic inference, allowing one to incorporate domain knowledge in modelling.

Max Entropy RL

In this case, maximizing reward is equivalent to inferring a trajectory with maximum probability. This in practice means maximize total reward, with an entropy regularization favoring more uniform policies, hence **maximum entropy RL**.

The **soft actor critic** algorithm is an off-policy actor critic algorithm whose objective is:

$$J^{SAC}(\theta) = \mathbb{E}_{p^{\infty}_{\pi_\theta}}[R(s,a) + \lambda \mathbb{H}(\pi_\theta(s))] \tag{645}$$

which outperforms off policy DDPG and on policy PPO by a wide margin on continuous control tasks.

Imitation Learning

Imitation Learning is when an agent does not observe rewards, but has access to a collection of trajectories generated by an expert policy. The goal is to imitate the expert, in the absence of reward.

A natural method is **behaviour cloning** which reduces IL to supervised learning. It interprets a policy as a classifier that maps sates to actions and finds a policy by minimizing the imitation error:

$$min_\pi \mathbb{E}_{p^{\infty}_{\pi_{exp}}}[D_{KL}(\pi_{exp}(s)|\pi(s)] \tag{646}$$

A challenge here is that this method does not consider the sequential nature of IL: future state distribution is not fixed but depends on earlier actions.

Inverse reinforcement learning infer a reward function that *explains* the observed expert trajectories and then computes a near optimal policy against this learned reward using any standard RL algorithms.

Since there are infinitely many reward function for which the expert policy is optimal, we can follow the *maximum entropy principle* and use an energy based probability model to capture how expert trajectories are generated:

$$p(\tau) \propto exp(\sum_{t=0}^{T-1} R_\theta(s_t, a_t)) \tag{647}$$

Causality