CSC412 Notes

Jenci Wei

Winter 2023

Contents

1 Introduction

Machine Learning

- **Supervised learning**: given input-output pairs $\left(x^{(i)}, y^{(i)}\right)$, learn the mapping f from inputs x to outputs y
- **Unsupervised learning**: given unlabelled data instances $x^{(i)}$, find relations among inputs, which can be used for making predictions/decisions
- **Semi-supervised learning**: given a limited amount of labelled data $\left(x^{(i)}, y^{(i)}\right)$, and a large amount of unlabelled data $x^{(i)}$
- **Reinforcement learning**: learning system receives a reward signal, tries to learn to maximize the reward signal
- All of the above aims to estimate distributions $\Pr(y \mid x), \Pr(x), \Pr(x, y)$ from data

Probabilistic Models

- Consider a random vector $X = (X_1, \ldots, X_d)$
- Want to model the relationship between these variables
- Probabilistic generative models: relate all varables by their joint probability distribution $\Pr(x) = \Pr(x_1, \ldots, x_d)$
- Suppose there is a true joint p_* which can be approximated by our model $\mathcal{P}$ (i.e. $p_* \approx p$ where $p \in \mathcal{P}$)
- Key questions
  - How we should specify a set of distributions $\mathcal{P}$
  - What it means for p to well approximate the true distribution p_*
  - How we can find a reasonable p efficiently
  - What modelling asumptions we should make (e.g. conditional independence)

Probabilistic Perspective on ML Tasks

- Random variables that we have
  - Input data x (high-dimensional)
  - Discrete outputs c (labels), or
  - Continuous outputs y
- If we have the joint probability over these random variables (e.g. $\Pr(x, y)$ or $\Pr(x, c)$), the we can perform the following:
  1. **Regression**
  $$\Pr(y \mid x) = \frac{\Pr(x, y)}{\Pr(x)} = \frac{\Pr(x, y)}{\int \Pr(x, y) dy}$$
  2. **Classification/Clustering**
  $$\Pr(c \mid x) = \frac{\Pr(x, c)}{\sum_c \Pr(x, c)}$$

Supervised Classification

- We observe pairs of input data and class labels

$$\left\{x^{(i)}, c^{(i)}\right\}_{i=1}^{N} \overset{\text{i.i.d.}}{\sim} \Pr(x, c)$$

- We would learn a distribution over class labels given new input data

$$\Pr(c \mid x) = \frac{\Pr(x, c)}{\sum_{c} \Pr(x, c)}$$

- **Discriminative models** deal with $\Pr(c \mid x)$
- **Generative models** deal with $\Pr(c, x)$

Observed vs. Unobserved Random Variables

- For supervised classification, we have a **supervised dataset** $\left\{x^{(i)}, c^{(i)}\right\}_{i=1}^{N} \sim \Pr(x, c)$, where the class labels are **observed**
- For unsupervised classification, the data is still generated from $\Pr(x, c)$ but we only observe $x^{(i)}$
- **Unsupervised dataset**: $\left\{x^{(i)}\right\}_{i=1}^{N} \sim \Pr(x) = \sum_{c} \Pr(x, c)$
  - We call an unobserved discrete class a *cluster*

Modelling Assumptions

- In order to learn p_* from data $\left\{x^{(i)}\right\}$, we make modelling assumptions:
  1. **IID data**: samples $x^{(i)}$ are i.i.d.
  2. **Parametrized distributions**: the distribution comes from a parametrized family $\mathcal{P} = \{\Pr(x \mid \theta) : \theta \in \Theta\}$
     - Reduces the complexity of our search space to the complexity of Θ
     - E.g. $\mathcal{P} = \{\Pr(x \mid \theta) = N(\theta, 1) : \theta \in \mathbb{R}\}$

Likelihood Function

- Let $x^{(i)} \sim \Pr(x \mid \theta_*)$ for $i = 1, \ldots, N$ be i.i.d random variables
- The joint of $\mathcal{D} = \left\{x^{(1)}, \ldots, x^{(N)}\right\}$ is $\Pr(\mathcal{D} \mid \theta_*) = \prod_{i} \Pr\left(x^{(i)} \mid \theta_*\right)$
- We observe data $\mathcal{D}$ and θ_* is unknown
- Likelihood function

$$\mathcal{L}(\theta; \mathcal{D}) = \Pr(\mathcal{D} \mid \theta) = \prod_{i} \Pr\left(x^{(i)} \mid \theta\right)$$

- Log-likelihood function

$$l(\theta; \mathcal{D}) = \log \mathcal{L}(\theta; \mathcal{D}) = \sum_{i} \log \Pr\left(x^{(i)} \mid \theta\right)$$

- If x is discrete, then $\mathcal{L}(\theta; \mathcal{D})$ is the probability of observing $\mathcal{D}$

Maximum Likelihood Estimation

- Want to estimate the true parameter θ_*

- Can pick parameter values which were most likely to have generated the data

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} l(\theta; \mathcal{D}) = \arg\max_{\theta} \mathcal{L}(\theta; \mathcal{D})$$

- The MLE estimator for a Bernoulli distribution is

$$\hat{\theta}_{\text{MLE}} = \frac{1}{N} \sum_{i=1}^{N} x^{(i)}$$

Sufficient Statistics

- A **sufficient statistic** is a statistic that conveys exactly the same information about the parameter as the entire data
- Fisher-Neyman Factorization Theorem: $T(x)$ is a sufficient statistics for the parameter θ in the parametric model $\Pr(x \mid \theta)$ iff

$$\Pr(x \mid \theta) = h(x) g_\theta(T(x))$$

  for some functions h (that does not depend on θ) and g_θ

Exponential Families

- Density of a member of exponential families is of the form

$$\Pr(x \mid \eta) = h(x) \exp\left(\eta^\top T(x) - A(\eta)\right)$$

  - $T(x)$ is a sufficient statistic
  - η is a natural parameter
  - $A(\eta)$ is the log-partition function
  - $h(x)$ is a carrying measure
- In the exponent, natural parameter interacts with the data only through the sufficient statistics
- E.g. (multivariate) Gaussian distribution, gamma, exponential, chi-squared, beta, Dirichlet, Poisson, geometric
- Example of 1-D Bernoulli distribution

$$\begin{aligned}
\Pr(x \mid \theta) &= \theta^x (1-\theta)^{1-x} \\
&= \exp\left(x \log \theta + (1-x) \log(1-\theta)\right) \\
&= \exp\left(x \log\left(\frac{\theta}{1-\theta}\right) + \log(1-\theta)\right)
\end{aligned}$$

  where

$$\begin{aligned}
T(x) &= x \\
\eta &= \log\left(\frac{\theta}{1-\theta}\right) \\
A(\eta) &= \log(1+e^\eta) \\
h(x) &= 1
\end{aligned}$$

Mean of Sufficient Statistics

- Moments of exponential families can be computed using the log-partition function
- Let $X \sim \Pr(x \mid \eta)$, then

$$\mathbb{E}[T(X)] = A'(\eta)$$

MLE for General Exponential Families

- The MLE for the natural parameters η of a general exponential family satisfies

$$A'\left(\hat{\eta}_{\text{MLE}}\right) = \frac{1}{N}\sum_{i=1}^{N} T\left(x^{(i)}\right)$$

  which may not have an explicit solution

2 Decision Theory

Decision Making

- Suppose we have a real-valued input vector x and a target (output) value c with joint probability distribution $\Pr(x, c)$
- Goal is to predict c given a new value for x
- The joint probability distribution $\Pr(x, c)$ provides a complete summary of uncertainties associated with these random variables
- Estimating $\Pr(x, c)$ from training data is an example of **inference**

Inference

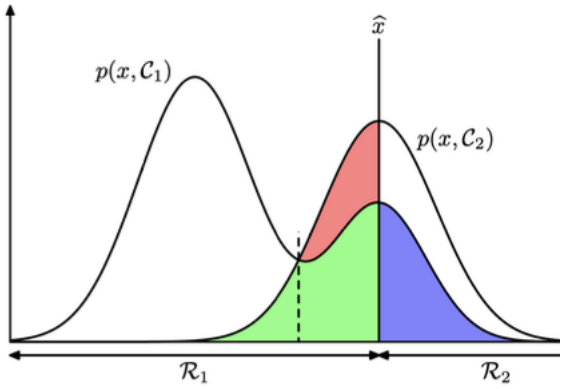
- Problem: assume we estimated the joint distribution $\Pr(x, \mathcal{C})$ using some ML method; in the end, we make a decision based on the probability of each target value

$$\Pr(\mathcal{C}_k \mid x) = \frac{\Pr(x \mid \mathcal{C}_k) \Pr(\mathcal{C}_k)}{\Pr(x)} \qquad \text{(Bayes rule)}$$

- We can minimize the expected number of mistakes (i.e. probability of assigning x to the wrong class)

Misclassification Rate

- Goal: make as few misclassifications as possible
- Divide the input space into regions $\mathcal{R}_k$ (decision regions) such that all points in $\mathcal{R}_k$ are assigned to class $\mathcal{C}_k$



  - Red + green regions: input belongs to $\mathcal{C}_2$ but is classified as $\mathcal{C}_1$
  - Blue region: input belongs to $\mathcal{C}_1$ but is classified as $\mathcal{C}_2$

$$\Pr(\text{mistake}) = \Pr(x \in \mathcal{R}_1, \mathcal{C}_2) + \Pr(x \in \mathcal{R}_2, \mathcal{C}_1)$$

  - Blue + green regions is always included in Pr(mistake)
  - Can reduce red area by moving the threshold to the left

- Misclassification error

$$\Pr(\text{mistake}) = \int_{\mathcal{R}_1} \Pr(x, \mathcal{C}_2) dx + \int_{\mathcal{R}_2} \Pr(x, \mathcal{C}_1) dx$$

- For a particular input x, if $\Pr(x, \mathcal{C}_1) > \Pr(x, \mathcal{C}_2)$, then we classify x as $\mathcal{C}_1$; otherwise, we classify x as $\mathcal{C}_2$

- To minimize misclassification, since $\Pr(x, \mathcal{C}_k) = \Pr(\mathcal{C}_k \mid x) \Pr(x)$, we assign x to the class for which the posterior probability $\Pr(\mathcal{C}_k \mid x)$ is largest

Expected Loss

- Want a **loss function** to measure the loss incurred by taking any of the available decisions
- Loss matrix: L_{kj} represents the true class being $\mathcal{C}_k$ but we assign it to $\mathcal{C}_j$
- Expected loss:

$$\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} \Pr(x, \mathcal{C}_k) dx$$

- Can choose the decision regions to minimize expected loss
- Rewriting the expected loss:

$$\begin{aligned} \mathbb{E}[L] &= \sum_j \int_{\mathcal{R}_j} \sum_k L_{kj} \Pr(x, \mathcal{C}_k) dx \\ &= \sum_j \int_{\mathcal{R}_j} g_j(x) dx \end{aligned}$$

  where

$$g_j(x) = \sum_k L_{kj} \Pr(x, \mathcal{C}_k)$$

- Thus minimizing $\mathbb{E}[L]$ is equivalent to choosing

$$\mathcal{R}_j = \{x : g_j(x) < g_i(x) \text{ for all } i \neq j\}$$

- Further simplifying using the fact $\Pr(x, \mathcal{C}_1) = \Pr(\mathcal{C}_1 \mid x) \Pr(x)$, we aim to find regions $\mathcal{R}_j$ such that the following is minimized:

$$\sum_k L_{kj} \Pr(\mathcal{C}_k \mid x)$$

  that is

$$\mathcal{R}_j = \left\{ x : \sum_k L_{kj} \Pr(\mathcal{C}_k \mid x) < \sum_k L_{ki} \Pr(\mathcal{C}_k \mid x) \text{ for all } i \neq j \right\}$$

Reject Option

- For the regions where we are uncertain about class membership, we don't have to make a decision

Loss Functions For Regression

- Consider an input/target setup (x, t) where $t \in \mathbb{R}$, and the joint density is $\Pr(x, t)$
- Instead of decision regions, we aim to find a regression function $y(x) \approx t$ which maps inputs to the outputs
- Consider the square loss function L:

$$L(y(x), t) = (y(x) - t)^2$$

- Want to minimize the expected loss

$$\mathbb{E}[L] = \iint L(y(x), t) \Pr(x, t) dx dt$$

- Can decompose expected loss into the sum of two *nonnegative* terms

$$\mathbb{E}[L] = \iint (y(x) - \mathbb{E}[t \mid x])^2 \Pr(x,t) dx dt + \iint (\mathbb{E}[t \mid x] - t)^2 \Pr(x,t) dx dt$$

  - The second term does not depend on $y(x)$, so it suffices to minimize the first term
  - Can make first term zero by choosing $y(x) = \mathbb{E}[t \mid x]$
  - The second term represents the intrinsic variability of the target data, and can be regarded as noise

3 Graphical Models

Joint Distributions

- The joint distribution of N random variables $(x_1, \ldots, x_N)$ is a general way to encode knowledge about a system
- Assume x_i are binary, then it requires $2^N - 1$ parameters to specify the joint distribution

$$\Pr(x_1, \ldots, x_N) = \prod_{j=1}^{N} \Pr(x_j \mid x_1, \ldots, x_{j-1})$$

Conditional Independence

- Assume there are N random variables $x_1, \ldots, x_N$
- For set $A \subset \{1, \ldots, N\}$, denote $x_A = \{x_i : i \in A\}$
- Assume A, B, C are disjoint, if random variables x_A, x_B are conditionally independent givenm x_C, then we write

$$x_A \perp x_B \mid x_C$$

- Equivalencies:

$$\begin{aligned} & x_A \perp x_B \mid x_C \\ \iff & \Pr(x_A, x_B \mid x_C) = \Pr(x_A \mid x_C) \Pr(x_B \mid x_C) \\ \iff & \Pr(x_A \mid x_B, x_C) = \Pr(x_A \mid x_C) \\ \iff & \Pr(x_B \mid x_A, x_C) = \Pr(x_B \mid x_C) \end{aligned}$$

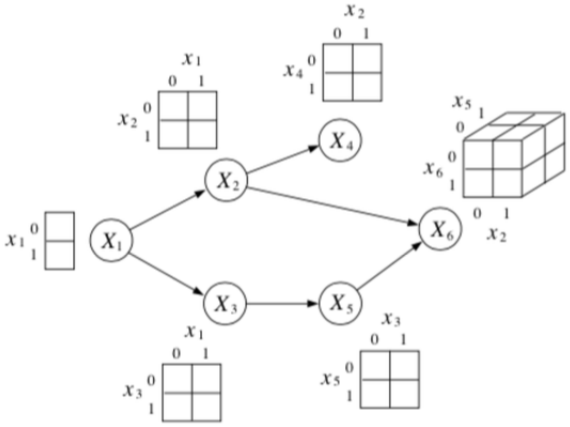
Directed Acyclic Graphical Models (Bayes' Nets)

- A directed acyclic graphical model (DAG) implies a factorization of the joint distribution
- Nodes represent variables
- Edges represent dependence
- Let parents(x_i) denote the set of nodes with edges pointing to x_i, then

$$\Pr(x_1, \ldots, x_N) = \prod_{i=1}^{N} \Pr(x_i \mid x_1, \ldots, x_{i-1}) = \prod_{i=1}^{N} \Pr(x_i \mid \text{parents}(x_i))$$

Conditional Probability Tables (CPT)

- Suppose each x_i is a binary random variable



- A 2×2 CPT requires 2 parameters
- Each CPT with K_i parents require 2^{K_i} parameters, which is $\approx N2^{\max K_i}$ parameters in total
- If we allow all possible dependencies (i.e. fully connected DAG), then we need $2^N - 1$ parameters
- DAGs reduce the computational burden of making inferences by introducing conditional independencies

Conditional Independence in DAGs

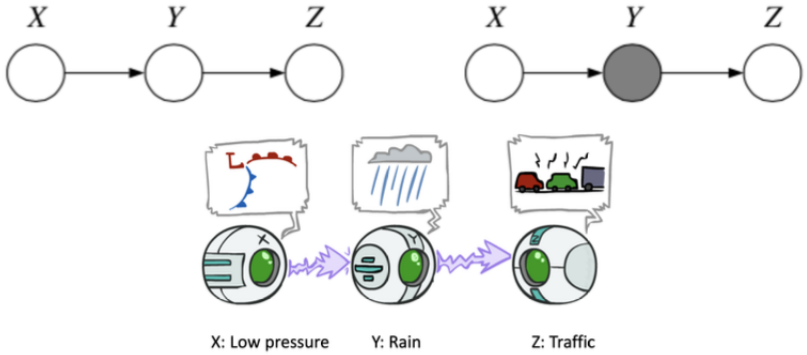
- **D-seperation** (directed-separation) is a notion of connectedness in DAGs in which two (sets of) variables may/may not be connected conditioned on a third (set of) variable(s)
- D-separation implies conditional independence and vice versa

DFS Algorithm of Checking Independence

- Let A, B, C be disjoint subsets of $\{1, \ldots, N\}$
- Cycle through each node in A, do a DFS to reach every node in B, and examine the path between them
- If all the paths have d-separated end points (i.e. conditinoally independent nodes), then

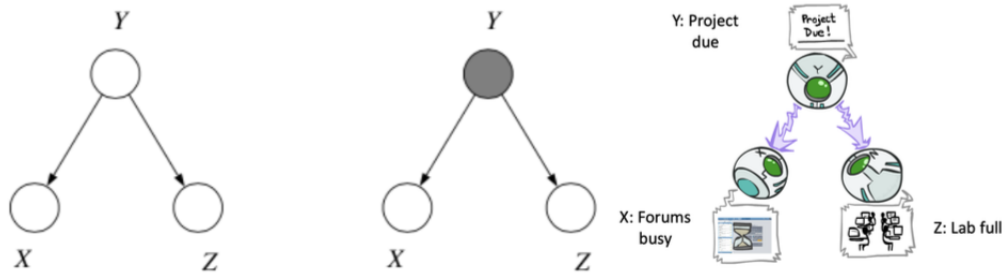
$$x_A \perp x_B \mid x_C$$

Causal Chain



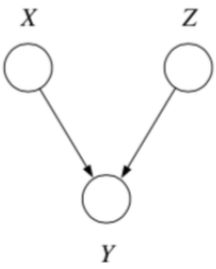
- X and Z d-separated given Y

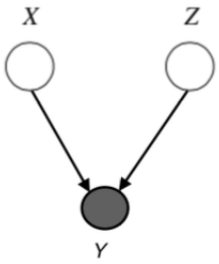
Common Cause



- X and Z d-separated given Y

Explaining Away (Common Effect)



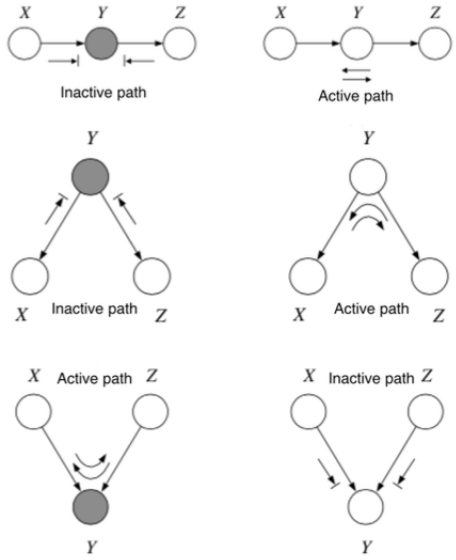


- X and Z **not d-separated** given Y

Bayes Ball Algorithm

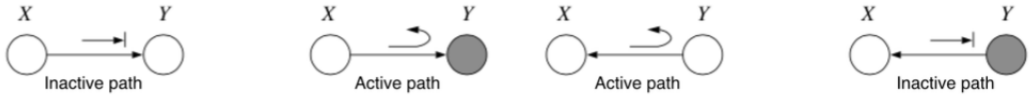
- An algorithm that determines conditional independence in a DAG
- To check if $x_A \perp x_B \mid x_C$, we need to check if every variable in A is d-separated from every variable in B conditioned on all variables in C
- Algorithm:
  1. Shade all nodes x_C (i.e. the observed)
  2. Place "balls" at each node in x_A
  3. Let the "balls" "bounce" around according to some rules
     - If any of the balls reach any of the nodes in x_B from x_A, then $x_A \not\perp x_B \mid x_C$
     - Otherwise $x_A \perp x_B \mid x_C$

Bayes Ball Rules



- Arrows: paths the balls can travel
- Arrows with bars: paths the balls cannot travel
- Balls can travel opposite to graph edge directions

- Pairs are boundary cases



4 Markov Random Fields

Limitations of DAG Models

- Each node is conditionally independent of its ancestors given its parents
- For some problems, it is not clear how to choose the edge directions in DAGMs
- **Markov blanket**: the set of nodes that renders a node conditionally independent of all the other nodes in the graph
  - Denoted mb, e.g. $mb(X)$

Markov Random Fields

- Undirected graphical models, aka Markov random fields (MRFs), are a set of random variables where the dependencies are described by an undirected graph
- A node represents a random variable
- An edge represents a probabilistic interation between the two nodes (as opposed to conditional dependence)
- Not all MRFs can be represented as DAGMs and vice versa

Dependencies in MRFs

1. **Global Markov Property (G)**: $x_A \perp x_B \mid x_C$ iff x_C separates x_A from x_B, i.e. there is no path in the graph between A and B that doesn't go through C
2. **Local Markov Property/Markov Blanket (L)**: the set of nodes that renders a node t conditionally independent of all the other nodes in the graph
   $$t \perp (\text{all} \backslash cl(t)) \mid mb(t)$$
   where $cl(t) = mb(t) \cup \{t\}$ is the closure of node t
3. **Pairwise (Markov) Property (P)**: the set of nodes that renders two nodes, s and t, conditionally independent of each other
   $$s \perp t \mid (\text{all} \backslash \{s, t\}) \iff (s, t) \notin E$$

Cliques

- A **clique** is a subset of nodes such that every two vertices in the subset are connected by an edge
  - **Maximal clique**: a clique that cannot be extended by including one more adjacent vertex
  - **Maximum clique**: a clique of the largest possible isze in a given graph

Distributions Induced by MRFs

- Let $x = (x_1, \ldots, x_m)$ be the set of all random variables in the graph
- No topological ordering in MRFs, so cannot use the chain rule to simplify $\Pr(x)$
- We associate potential functions with each maximal clique, i.e. given a maximal clique c, the potential function (or factor)
  $$\psi_c(x_c \mid \theta_c)$$
  is a nonnegative function, where x_c is the subset of variables in c and θ_c is some parameter

- The joint distribution is proportional to the product of clique potentials

$$\Pr(x) \propto \prod_{c \in \mathcal{C}} \psi_c(x_c \mid \theta_c)$$

where $\mathcal{C}$ is the set of all maximal cliques

- Any distribution whose conditional inddependencies are represented with an MRF can be represented this way
- A distribution $\Pr(x) > 0$ satisfies the conditional independence properties of an undirected ggraph iff $\Pr(x)$ can be represented as a product of factors, one per maximal clique, i.e.

$$\Pr(x \mid \theta) = \frac{1}{Z(\theta)} \prod_{c \in \mathcal{C}} \psi_c(x_c \mid \theta_c)$$

where $\mathcal{C}$ is the set of all maximal cliques of G, and $Z(\theta)$ is the **partition function**, defined as

$$Z(\theta) = \sum_x \prod_{c \in \mathcal{C}} \psi_c(x_c \mid \theta_c)$$

MRFs as Exponential Families

- Can write this as an exponential family:

$$\Pr(x \mid \theta) = \exp\left\{ \sum_{c \in \mathcal{C}} \log \phi_c(x_c \mid \theta_c) - \log(Z(\theta)) \right\}$$

- If the potentials have a log-linear form (model assumption)

$$\log \psi_c(x_c \mid \theta_c) = \theta_c^\top \phi_c(x_c)$$

then

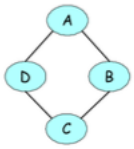
$$\Pr(x \mid \theta) = \exp\left\{ \sum_{c \in \mathcal{C}} \theta_c^\top \phi_c(x_c) - \log Z(\theta) \right\}$$

- The expectation of the cth feature is

$$\mathbb{E}[\phi_c(x_c)] = \frac{\partial \log Z(\theta)}{\partial \theta_c}$$

Representing Potentials

- If the variables are discrete, then we can represent the potential functions as tables of (nonnegative) numbers
- E.g.



$\psi_{AB}[A, B]$

a^0	b^0	30
a^0	b^1	5
a^1	b^0	1
a^1	b^1	10

$\psi_{BC}[B, C]$

b^0	c^0	100
b^0	c^1	1
b^1	c^0	1
b^1	c^1	100

$\psi_{CD}[C, D]$

c^0	d^0	1
c^0	d^1	100
c^1	d^0	100
c^1	d^1	1

$\psi_{AD}[D, A]$

d^0	a^0	100
d^0	a^1	1
d^1	a^0	1
d^1	a^1	100

$$\Pr(A, B, C, D) = \frac{1}{Z} \psi_{A,B}(A, B) \psi_{B,C}(B, C) \psi_{C,D}(C, D) \psi_{A,D}(A, D)$$

- The potentials are not probabilities, they encode relative affinities between the different assignments (i.e. how much more likely is an assignment to be true than another)

Ising Model

- The Ising model is an MRF that is used to model magnets
- The node variables are spins, i.e. $x_s \in \{-1, +1\}$
- Define the pairwise clique potentials as

$$\phi_{st}(x_s, x_t) = \begin{bmatrix} e^{W_{st}} & e^{-W_{st}} \\ e^{-W_{st}} & e^{W_{st}} \end{bmatrix}$$

  where W_{st} is the coupling strength between nodes s and t
- Since for $(x_s, x_t) = (1, -1)$ we have $\psi_{st}(1, -1) = e^{-W_{st}}$, we can write the pairwise potential as

$$\psi_{st}(x_s, x_t) = e^{x_s x_t W_{st}}$$

- If two nodes are connected, then we set $W_{st} = J$, otherwise $W_{st} = 0$
  - It is more likely to have the same neighbouring spins
- Since pairwise potential only tells us about the pairwise behaviours, we also want to add node potentials:

$$\psi_s(x_s) = e^{b_s x_s}$$

- The overall distribution becomes

$$\Pr(x) \propto \prod_{s \sim t} \psi_{st}(x_s, x_t) \prod_s \psi_s(x_s) = \exp\left\{ J \sum_{s \sim t} x_s x_t + \sum_s b_s x_s \right\}$$

- If $x_s = x_t$, i.e. the neighbouring spins are the same, then the likelihood is larger

5 Exact Inference

Inference as Conditional Distribution

- We can explore inference in probabilistic graphical models (PGMs)
  - x_E is the observed evidence
  - x_F is the unobserved variable we want to infer
  - $x_R = \{x_F, x_E\}$ is the remaining variables, extraneous to query
- For inference, we focus on computing the conditional probability distribution

$$\Pr(x_F \mid x_E) = \frac{\Pr(x_F, x_E)}{\Pr(x_E)} = \frac{\Pr(x_F, x_E)}{\sum_{x_F} \Pr(x_F, x_E)}$$

- We can marginalize out the extraneous variables:

$$\Pr(x_F, x_E) = \sum_{x_R} \Pr(x_F, x_E, x_R)$$

Variable Elimination

- Order which variables are marginalized affects the computational cost
- **Variable elimination algorithm**
  - A simple and general **exact inference** algorithm in any probabilistic graphical model (DAGMs or MRFs)
  - Has computational complexity that depends on the graph structure of the model
  - Can use dynamic programming to avoid enumerating all variable assignments
- E.g. consider a chain $A \to B \to C \to D$
  - Want to compute the marginal $\Pr(D)$
  - Naive approach:

$$\Pr(D) = \sum_{A,B,C} \Pr(A, B, C, D) = \sum_{C} \sum_{B} \sum_{A} \Pr(A) \Pr(B \mid A) \Pr(C \mid B) \Pr(D \mid C)$$

    whose cost is $\mathcal{O}(k^{n=4})$
  - Can choose an *elimination ordering*

$$\Pr(D) = \sum_{C,B,A} \Pr(A, B, C, D) = \sum_{C} \Pr(D \mid C) \sum_{B} \Pr(C \mid B) \sum_{A} \Pr(B \mid A) \Pr(A)$$

    whose cost is $\mathcal{O}(nk^2)$

Best Elimination Ordering

- Complexity of variable elimination depends on the elimination ordering
- Finding the best elimination ordering is NP-hard

Intermediate Factors

- In general, eliminating does not produce a valid marginal or conditional distribution of the graphical model

- E.g. we want to marginalize over X

$$\begin{aligned}\Pr(A,B,C) &= \sum_X \Pr(X)\Pr(A \mid X)\Pr(B \mid A)\Pr(C \mid B,X) \\ &= \Pr(B \mid A)\sum_X \Pr(X)\Pr(A \mid X)\Pr(C \mid B,X)\end{aligned}$$

  - The sum does not correspond to a valid distribution because it is unnormalized

- We introduce **factors** which are not necessarily normalized distributions, but describe the local relationship between random variables

- E.g.

$$\begin{aligned}\Pr(A,B,C) &= \sum_X \Pr(X)\Pr(A \mid X)\Pr(B \mid A)\Pr(C \mid B,X) \\ &= \sum_X \phi_1(X)\phi_2(A,X)\phi_3(A,B)\phi_4(X,B,C) \\ &= \phi_3(A,B)\sum_X \phi_1(X)\phi_2(A,X)\phi_4(X,B,C) \\ &= \phi_3(A,B)\tau(A,B,C)\end{aligned}$$

  - Marginalizing over X we introduce a new factor τ

Sum-Product Inference

- Computing $\Pr(x_F \mid x_E)$ is given by the **sum-product algorithm**

$$\Pr(x_F \mid x_E) \propto \tau(x_F, x_E) = \sum_{x_R} \prod_{\phi_c \in \Phi} \phi_c(x_c)$$

  where Φ is a set of potentials or factors

- For DAGMs, Φ is given by the conditional probability distributions for all variables

$$\Phi = \{\phi_{x_i}\}_{i=1}^N = \{\Pr(x_i \mid \text{parents}(x_i))\}_{i=1}^N$$

  where the sum is over the set $x_R = x - x_F - x_E$

- For MRFs, Φ is given by the set of unnormalized potentials, therefore we must normalize the resulting $\tau(t)$ by $\sum_y \tau(y)$

Complexity of Variable Elimination Ordering

$$\mathcal{O}\left(mk^{N_{\max}}\right)$$

- m is the number of initial factors
- k is the number of states each random variable takes (assumed equal)
- N_i is the number of random variables inside each sum $\sum_i$ (counting i itself)
- $N_{\max}$ is the number of variables inside the largest sum

6 Message Passing

Latent Variables

- Assume variable z is unobserved
- If we never condition on z, then we can integrate it out
- In certain cases we are interested in the latent variables themselves

TrueSkill Latent Variable Model

- Player ranking system for competitive games
- Goal is to infer the skill of a set of players in a competitive game, based on observing who beats who
- Each player has a fixed level of skill, denoted z_i
- We initially don't know anything about anyone's skill, but we assume everyone's skill is independent (e.g. an independent Gaussian prior)
- We never observe the players' skills directly, which makes this a latent variable model
- Instead, we observe the outcome of a series of matches between different players
- For each game, the probability that player i beats player j is given by

$$\Pr(i \text{ beats } j) = \sigma(z_i - z_j)$$

- Can write the entire joint likelihood of a set of players and games as

$$\Pr(z_1, \ldots, z_N, \text{game 1}, \ldots, \text{game T}) = \left[\prod_{i=1}^{N} \Pr(z_i)\right] \left[\prod_{\text{games}} \Pr(i \text{ beats } j \mid z_i, z_j)\right]$$

Posterior

- Given the outcome of some matches, the players' skills are no longer independent, even if they've never played each other
- Computing the posterior over two players' skills require integrating over all the other players' skills:

$$\Pr(z_1, z_2 \mid \text{game 1}, \ldots, \text{game T}) = \int \cdots \int \Pr(z_1, z_2, z_3, \ldots, z_N \mid x) dz_3 \cdots dz_N$$

- Message passing can be used to compute posteriors

Variable Elimination Order and Trees

- We can do exact inference by variable elimination
- Computational cost is determined by the graph structure, and the elimination ordering
- Determining the optimal elimination ordering is hard – even if we do, the resulting marginalization might be unreasonably costly
- For trees, any elimination ordering that goes from the leaves inwards towards any root will be optimal

Inference in Trees

- A graph is $G = (V, E)$ where V is the set of vertices and E is the set of edges

- For $i, j \in V$, we have $(i, j) \in E$ if there is an edge between the nodes i and j
- For a node in graph $i \in V$, $N(i)$ denotes the neighbours of i, i.e. $N(i) = \{j : (i, j) \in E\}$
- Shaded nodes are observed, and denoted by $\overline{x}$
- Joint distribution:

$$\Pr(x_{1:n}) = \frac{1}{Z} \prod_{i \in V} \psi(x_i) \prod_{(i,j) \in E} \psi_{ij}(x_i, x_j)$$

Message Passing on Trees

- We perform variable elimination from leaves to root, which is the sum product algorithm to compute all marginals
- Belief propagation is a message-passing between neighbouring vertices of the graph
- The message sent from variable j to $i \in N(j)$ is

$$m_{j \to i}(x_i) = \sum_{x_j} \psi_j(x_j) \psi_{ij}(x_i, x_j) \prod_{k \in N(j) \setminus i} m_{k \to j}(x_j)$$

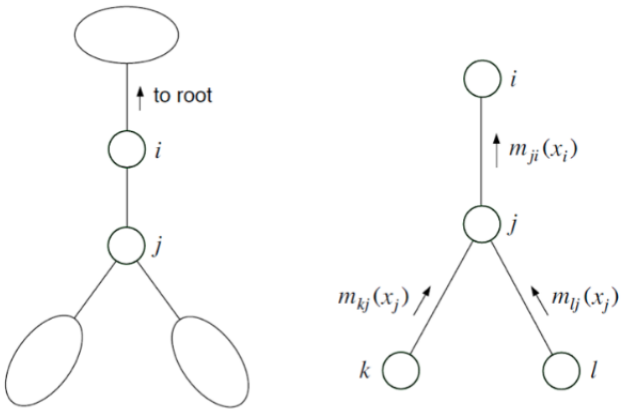
- If x_j is observed, then the message is

$$m_{j \to i}(x_i) = \psi_j(\overline{x}_j) \psi_{ij}(x_i, \overline{x}_j) \prod_{k \in N(j) \setminus i} m_{k \to j}(\overline{x}_j)$$

- Each message $m_{j \to i}(x_i)$ is a vector with one value for each state of x_i
- Once the message passing stage is complete, we can compute our beliefs as

$$b(x_i) \propto \psi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i)$$

- Once normalized, beliefs are the marginals we want to compute



Belief Propagation on Trees

- Algorithm
  1. Choose root r arbitrarily
  2. Pass messages from leaves to r
  3. Pass messages from r to leaves

4. Compute beliefs (marginals)

- Compute beliefs in two steps
  1. Compute unnormalized beliefs
  $$\tilde{b}(x_i) = \phi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i)$$
  2. Normalize beliefs
  $$b(x_i) = \frac{\tilde{b}(x_i)}{\sum_{x_i} \tilde{b}(x_i)}$$

Loopy Belief Propagation

- The graph (MRF) might not be a tree and have cycles
- **Loopy belief propagation**: keep passing messages until convergence
- We won't get the exact marginals, but an approximation
- Algorithm
  1. Initialize all messages uniformly
  $$m_{i \to j}(x_j) = \begin{bmatrix} 1/k \\ \vdots \\ 1/k \end{bmatrix}$$
  where k is the number of states x_j can take
  2. Keep running BP updates until it "converges"
  $$m_{j \to i}(x_i) = \sum_{x_j} \psi_j(x_j) \psi_{ij}(x_i, x_j) \prod_{k \in N(j) \neq i} m_{k \to j}(x_j)$$
  and normalize for stability
  3. If it does not converge, it's fine
  4. Compute beliefs
  $$b(x_i) \propto \psi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i)$$
- This algorithm is useful in practice despite lack of theoretical guarantee

Sum-Product vs. Max-Product

- The algorithm above is the **sum-product BP** and approximately computes the *marginals* at each node
- For MAP inference, we maximize over x_j instead of summing over them, which is **max-product BP**
- BP updates take the form
$$m_{j \to i}(x_i) = \max_{x_j} \psi_j(x_j) \psi_{ij}(x_i, x_j) \prod_{k \in N(j) \neq i} m_{k \to j}(x_j)$$
- After BP algorithm converges, the beliefs are **max-marginals**
$$b(x_i) \propto \psi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i)$$
- MAP inference:
$$\hat{x}_i = \arg\max_{x_i} b(x_i)$$

7 Sampling

Sampling

- A sample from a distribution $p(x)$ is a single realization x whose probability distribution is $p(x)$
  - x can be high-dimensional or real-valued
- Assume the density from which we wish to draw samples, $p(x)$, can be evaluated to within a multiplicative constant
  - We can evaluate a function $\tilde{p}(x)$ such that

$$p(x) = \frac{\tilde{p}(x)}{Z}$$

Ancestral Sampling

- Given a DAG
- We have the ability to sample from each of its factors given its parents
- We can sample from the joint distribution over all the nodes by **ancestral sampling**, which samples in a topological order
- At each step, sample from any conditional distribution that we haven't visited yet, whose parents have all been sampled
- E.g. in a chain we start from the head, and in a tree we start from the root

Objectives of Sampling

- We will use Monte Carlo methods to solve the following problems:
  1. Generate smaples $\left\{x^{(r)}\right\}_{r=1}^{R}$ from a given probability distribution $p(x)$
  2. Estimate expectations of functions, $\phi(x)$, under this distribution $p(x)$

$$\Phi = \mathbb{E}_{x\sim p(x)}[\phi(x)] = \int \phi(x)p(x)dx$$

  - ϕ is called a *test function*

Examples of Test Functions

- The mean of a function f under $p(x)$ by finding the expectation of the function $\phi_1(x) = f(x)$
- The variance of f under $p(x)$ by finding the expectations of functions $\phi_1(x) = f(x)$ and $\phi_2(x) = f(x)^2$

$$\begin{aligned} \phi_1(x) = f(x) &\implies \Phi_1 = \mathbb{E}_{x\sim p(x)}[\phi_1(x)] \\ \phi_2(x) = f(x)^2 &\implies \Phi_2 = \mathbb{E}_{x\sim p(x)}[\phi_2(x)] \\ &\implies \mathrm{Var}[f(x)] = \Phi_2 - \Phi_1^2 \end{aligned}$$

Estimation Problem

- **Simple Monte Carlo**: given $\left\{x^{(r)}\right\}_{r=1}^{R} \sim p(x)$, we can estimate the expectation $\mathbb{E}_{x\sim p(x)}[\phi(x)]$ using the estimator $\hat{\Phi}$

$$\Phi = \mathbb{E}_{x\sim p(x)}[\phi(x)] \approx \frac{1}{R}\sum_{r=1} \phi\left(x^{(r)}\right) = \hat{\Phi}$$

- $\hat{\Phi}$ is a consistent estimator by the Law of Large Numbers (LLN)

Properties of Monte Carlo Estimation

- **Unbiasedness**: if the vectors $\left\{x^{(r)}\right\}_{r=1}^{R}$ are generated independently from $p(x)$, then

$$\mathbb{E}[\hat{\Phi}] = \Phi$$

- **Variance**: as the number of samples of R increases, the variance of $\hat{\Phi}$ will decrease with rate $\frac{1}{R}$

$$\text{Var}[\hat{\Phi}] = \frac{1}{R}\text{Var}[\phi(x)]$$

  - Accuracy of the Monte Carlo estimate depends on the variance of ϕ

Sampling Problem

- Assume we know the density $p(x)$ up to a multiplicative constant

$$p(x) = \frac{\tilde{p}(x)}{Z}$$

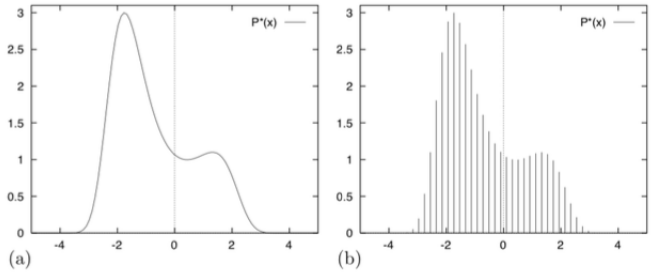
- Two difficulties
  1. We do not generally know the normalizing constant Z, computing

$$Z = \int \tilde{p}(x)dx$$

     requires computing a high-dimensional integral
  2. Even if we did know Z, it is still challenging to draw samples from $p(x)$, especially in high-dimensional spaces

Lattice Discretization

- We want to draw samples from the density $p(x)$



- We could discretize the variable x and sample from the discrete distribution
- If we have 50 uniformly spaced points in one dimension, then we would need a total of 50^D points, which is exponential in dimension and very costly

Analogy

- Task is to draw random water samples from a lake and find the average plankton concentration
  - $\tilde{p}(\boldsymbol{x})$ is the depth of the lake at $\boldsymbol{x} = (x, y)$
  - $\phi(\boldsymbol{x})$ is the plankton concentration as a function of $\boldsymbol{x}$
  - $Z = \int \tilde{p}(\boldsymbol{x})d\boldsymbol{x}$ is the volume of the lake

– The average concentration of plankton is

$$\Phi = \frac{1}{Z} \int \phi(\boldsymbol{x}) \tilde{p}(\boldsymbol{x}) d\boldsymbol{x}$$

- Can take the boat to any desired location $\boldsymbol{x}$ on the lake and measure the depth $\tilde{p}(\boldsymbol{x})$ and plankton concentration $\phi(\boldsymbol{x})$ at that point
- Problems
  1. Draw water samples at random such taht each sample is equally likely to come from any point within the lake
  2. Find the average plankton concentration

- We don't know the depth $\tilde{p}(\boldsymbol{x})$
- To correctly estimate Φ, we must implicitly discover the canyons and find their volume relative to the rest of the lake

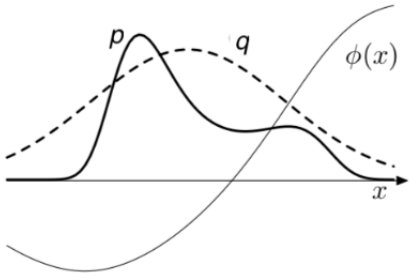
Importance Sampling

- **Importance sampling** is the method for estimating the expectation of a function $\phi(x)$
- The density from which we wish to draw samples, $p(x)$ can be evaluted up to a normalizing constant $\tilde{p}(x)$, where

$$p(x) = \frac{\tilde{p}(x)}{Z}$$

- There is a simpler density $q(x)$ from which it is easy to sample and easy to evaluate up to a normalizing constant (i.e. $\tilde{q}(x)$)

$$q(x) = \frac{\tilde{q}(x)}{Z_q}$$



- In importance sampling, we generate R samples from $q(x)$

$$\left\{x^{(r)}\right\}_{r=1}^{R} \sim q(x)$$

- If these points were samples from $p(x)$ then we could estimate Φ by

$$\Phi = \mathbb{E}_{x\sim p(x)}[\phi(x)] \approx \frac{1}{R}\sum_{r=1}\phi\left(x^{(r)}\right) = \hat{\Phi}$$

  i.e. we could use a simple Monte Carlo estimator

- Since we sampled from q, we need some corrections
- Values of x where $q(x) > p(x)$ will be overrepresented in this estimator, and values of x where $q(x) < p(x)$ will be underrepresented
- We can introduce weights

$$\tilde{w}_r = \frac{\tilde{p}\left(x^{(r)}\right)}{\tilde{q}\left(x^{(r)}\right)}$$

- Notice that

$$\frac{1}{R}\sum_{r=1}^{R}\tilde{w}_r \approx \mathbb{E}_{x\sim q(x)}\left[\frac{\tilde{p}(x)}{\tilde{q}(x)}\right] = \int\frac{\tilde{p}(x)}{\tilde{q}(x)}q(x)dx = \frac{Z}{Z_q} \tag{7.1}$$

- Rewriting our estimator under q:

$$\begin{aligned}
\Phi &= \int\phi(x)p(x)dx \\
&= \int\phi(x)\frac{p(x)}{q(x)}q(x)dx \\
&\approx \frac{1}{R}\sum_{r=1}^{R}\phi\left(x^{(r)}\right)\frac{p\left(x^{(r)}\right)}{q\left(x^{(r)}\right)} \qquad \text{From simple Monte Carlo} \\
&= \frac{Z_q}{Z}\frac{1}{R}\sum_{r=1}^{R}\phi\left(x^{(r)}\right)\cdot\frac{\tilde{p}\left(x^{(r)}\right)}{\tilde{q}\left(x^{(r)}\right)} \\
&= \frac{Z_q}{Z}\frac{1}{R}\sum_{r=1}^{R}\phi\left(x^{(r)}\right)\cdot\tilde{w}_r \\
&\approx \frac{\frac{1}{R}\sum_{r=1}^{R}\phi\left(x^{(r)}\right)\cdot\tilde{w}_r}{\frac{1}{R}\sum_{r=1}^{R}\tilde{w}_r} \qquad \text{From (7.1)} \\
&= \sum_{r=1}^{R}\phi\left(x^{(r)}\right)\cdot w_r \\
&= \hat{\Phi}_{iw}
\end{aligned}$$

  where

$$w_r = \frac{\tilde{w}_r}{\sum_{r=1}^{R}\tilde{w}_r}$$

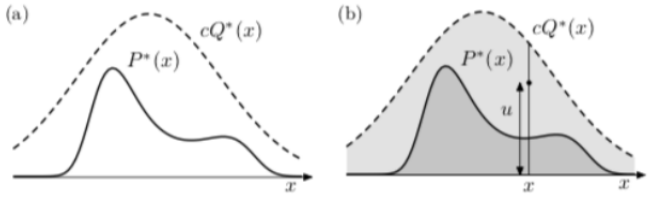
  and $\hat{\Phi}_{iw}$ is the importance weighted estimator

Rejection Sampling

- We want expectations under $p(x) = \tilde{p}(x)/Z$, which is a very complicated one-dimensional density

- Assume that we ahve a simpler proposal density $q(x)$ which we can evaluate (within a multiplicative factor Z_q, as before), and from which we can generate samples
- Further assume that we know the value of a constant c such that

$$c\tilde{q}(x) > \tilde{p}(x) \quad \forall x$$



- Algorithm
  1. Generate 2 random numbers x and u
     (a) x is generated from $q(x)$
     (b) u is generated uniformly from the interval $[0, c\tilde{q}(x)]$
  2. Evaluate $\tilde{p}(x)$ and accept or reject the sample x by comparing the value of u with the value of $\tilde{p}(x)$
     (a) If $u > \tilde{p}(x)$, then x is rejected
     (b) Otherwise x is accepted; x is added to our set of samples $\left\{x^{(r)}\right\}$ and the value of u is discarded
- Compact representation:
  1. $x \sim q(x)$
  2. $u \mid x \sim \text{Unif}[0, c\tilde{q}(x)]$
  3. x accepted if $u \leq \tilde{p}(x)$

Proof. (Rejection sampling works) For any set A:

$$\Pr_{x\sim p}(x \in A) = \int_A p(x)dx = \int \mathbb{1}\{x \in A\}\, p(x)dx = \mathbb{E}_{x\sim p}[\mathbb{1}\{x \in A\}] \tag{7.2}$$

Then

$$\begin{aligned}
\Pr_{x\sim q}(x \in A \mid u \leq \tilde{p}(x)) &= \frac{\Pr_{x\sim q}(x \in A, u \leq \tilde{p}(x))}{\mathbb{E}_{x\sim q}[\Pr(u \leq \tilde{p}(x) \mid x)]} \\
&= \frac{\mathbb{E}_{x\sim q}\mathbb{1}\{x \in A\}\Pr(u \leq \tilde{p}(x) \mid x)}{\mathbb{E}_{x\sim q}\left[\frac{\tilde{p}(x)}{c\tilde{q}(x)}\right]} \\
&= \frac{\mathbb{E}_{x\sim q}\left[\mathbb{1}\{x \in A\}\frac{\tilde{p}(x)}{c\tilde{q}(x)}\right]}{\frac{Z_p}{cZ_q}} \\
&= \frac{\Pr_{x\sim p}(x \in A)\frac{Z_p}{cZ_q}}{\frac{Z_p}{cZ_q}} \qquad \text{From (7.2)} \\
&= \Pr_{x\sim p}(x \in A)
\end{aligned}$$

□

Rejection Sampling in Many Dimensions

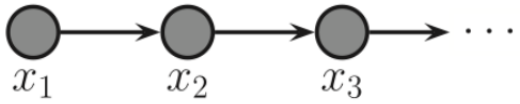
- In high-dimensional problems, the requirement that $c\tilde{q}(x) \geq \tilde{p}(x)$ forces c to be large to the point where acceptances are rare
- Finding such a value of c may be diffifult, since we don't know where the modes of $\tilde{p}$ are located nor how high they are
- In general c grows exponentially with the dimensionality, so the acceptance rate is expected to be exponentially small in dimension

$$\text{acceptance rate} = \frac{\text{area under } \tilde{p}}{\text{area under } c\tilde{q}} = \frac{1}{Z}$$

Sequential Data

- When modelling the data, we generally assume iid, but this is not always the case
- **Sequential data** are common:
  - Time-series modelling (e.g. stock prices)
  - Ordered data (e.g. textual data)
- The joint factorization via the chain rule becomes intractable for high-dimensional data, since each factor requires exponentially many parameters to specify

Markov Chains



- We make the **first-order Markov chain** assumption:

$$\Pr(x_t \mid x_{1:t-1}) = \Pr(x_t \mid x_{t-1})$$

- This assumption simplifies the factors in the joint distribution:

$$\Pr(x_{1:T}) = \prod_{t=1}^{T} \Pr(x_t \mid x_{t-1})$$

Stationary Markov Chains

- **Stationary (homogeneous) Markov chain**: the distribution generating data does not change through time

$$\Pr(x_{t+1} = y \mid x_t = x) = \Pr(x_{t+2} = y \mid x_{t+1} = x) \quad \forall t$$

- **Non-stationary Markov chain**: the transition probabilities $\Pr(x_{t+1} = y \mid x_t = x)$ depend on the time t
- We will focus on stationary Markov chains

Higher-Order Markov Chains

- The first-order assumption may be restrictive (e.g. when modelling natural language, there are long-term dependencies)
- Can generalize to high-order dependence

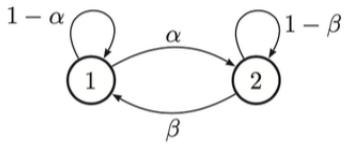
- Second order:
$$\Pr(x_t \mid x_{1:t-1}) = \Pr(x_t \mid x_{t-1}, x_{t-2})$$
- mth order:
$$\Pr(x_t \mid x_{1:t-1}) = \Pr(x_t \mid x_{t-1:t-m})$$

Transition Matrix

- When x_t is discrete (e.g. 1-of-K), the conditional distribution $\Pr(x_t \mid x_{t-1})$ can be written as a $K \times K$ matrix
- The **transition matrix** (or stochastic matrix) $A \in \mathbb{R}^{K \times K}$ satisfies
$$A_{ij} = \Pr(x_t = j \mid x_{t-1} = i)$$
- Note that
$$\Pr(x_t = j) = \sum_i \Pr(x_t = j \mid x_{t-1} = i) \Pr(x_{t-1} = i) = \sum_i A_{ij} \Pr(x_{t-1} = i)$$
- Each row of the matrix sums to 1, i.e.
$$\sum_j A_{ij} = 1$$

State Transition Diagram

- The transition matrix A specifies the probability of going from state i to state j
- Can visualize Markov chains via a directed graph, where
  - Nodes represent states
  - Arrows represent legal transitions (i.e. nonzero elements of A)



- Such diagram is called a **state transition diagram**
- The weights associated with the arcs are the probabilities
- The transition matrix for the above 2-state chain is
$$A = \begin{bmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{bmatrix}$$

Chapman-Kolmogorov Equations

- The n-step transition matrix $A(n)$ is defined as
$$A_{ij}(n) = \Pr(x_{t+n} = j \mid x_t = i)$$
which is the probability of getting from i to j in exactly n steps
- Notice that $A(1) = A$

- **Chapman-Kolmogorov** equations state that

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

  The probability of getting from i to j in $m+n$ steps is the sum of the probabilities of getting from i to k in m steps, and then from k to j in n steps, summed up for all k

- Therefore

$$A(n) = A^n$$

Stationary Distribution of a Markov Chain

- We are often interested in the long term distribution over states, which is the **stationary distribution** of the chain
- Let A be the transition matrix where

$$A_{ij} = \Pr(x_{t+1} = j \mid x_t = i)$$

- Let

$$\pi_t(j) = \Pr(x_t = j)$$

  be the probability of being in state j at time t

- The initial distribution is given by $\pi_0 \in \mathbb{R}^K$ and

$$\pi_1(j) = \sum_{i=1}^{K} \Pr(x_1 = j \mid x_0 = i)\pi_0(i) = \sum_{i=1}^{K} A_{ij}\pi_0(i) = \sum_{i=1}^{K} A_{ji}^{\top}\pi_0(i)$$

- Assume that π_t is a *row vector* with entries $\pi_t(j)$. This vector is the distribution of x_t, i.e. $\pi_t(j) = \Pr(x_t = j)$
- Then

$$\pi_1 = \pi_0 A \qquad \text{or more generally} \qquad \pi_t = \pi_0 A^t$$

- Updating π infinitely many times, the distribution of x_t may converge, i.e.

$$\pi = \pi A$$

  then we have reached the stationary distribution (aka the invariant distribution) of the Markov chain

- We can find this stationary distribution of a Markov chain by solving the eigenvector equation

$$A^{\top} v = v \qquad \text{and set} \qquad \pi = v$$

  where v is the eigenvector of $A^{\top}$ with eigenvalue 1

- Need to normalize
- The stationary distribution may not be unique

Detailed Balance Equations

- Markov chain is called:
  - **Irreducible** if we can get from any state to any other state
  - **Regular** if A^n has all positive entries for some n

– **Time reversible** if there exists a distribution π such that

$$\pi_i A_{ij} = \pi_j A_{ji} \qquad \forall i, j$$

called **detailed balance equations**, which means that going two ways between states is equally likely

Theorem 7.1. *If a Markov chain with transition matrix A is regular and satisfies detailed balance w.r.t. distribution π, then π is a stationary distribution*

Proof. WTS $A^\top \pi = \pi$, or equivalently

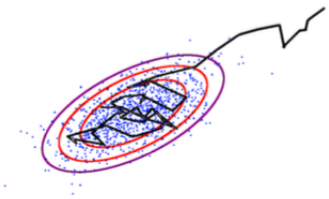
$$\sum_{i=1}^{K} \pi_i A_{ij} = \pi_j$$

For every $j = 1, \ldots, K$, we have

$$\begin{aligned} \sum_{i=1}^{K} \pi_i A_{ij} &= \sum_{i=1}^{K} \pi_j A_{ji} \qquad \text{By detailed balance} \\ &= \pi_j \sum_{i=1}^{K} A_{ji} \\ &= \pi_j \end{aligned}$$

□

Markov Chain Monte Carlo (MCMC)



- In contrast to rejection sampling where the accepted points $\left\{x^{(t)}\right\}$ are independent, MCMC methods generate a dependent sequence
- Each sample $x^{(t)}$ has a probability distribution that depends on the previous value $x^{(t-1)}$
- MCMC methods need to be run for a time in order to generate sapmles that are from the target distribution p
- We can still do Monte Carlo estimation for large enough T to estimate the mean of a test function ϕ:

$$\mathbb{E}_{x \sim p}[\phi(x)] \approx \frac{1}{T} \sum_{t=1}^{T} f\left(x^{(t)}\right)$$

- We might want to discard some initial samples

Metropolis-Hastings

- Importance and rejection sampling work only if the proposal density $q(x)$ is similar to $p(x)$
- It is hard to find such q in high dimensions
- The Metropolis-Hastings algorithm makes use of a proposal density q that depends on the current state $x^{(t)}$

- The density $q\left(x \mid x^{(t)}\right)$ might be a simple distribution such as a Gaussian centered at $x^{(t)}$, but can be any density from which we can draw samples
- It is *not* necessary that $q\left(x \mid x^{(t)}\right)$ looks similar to $p(x)$
- Assume we can evaluate $\tilde{p}(x)$ for any x
- Algorithm
  1. A tentative new state x' is generated form the proposal density $q\left(x' \mid x^{(t)}\right)$
  2. We accept the new state with probability

$$a\left(x' \mid x^{(t)}\right) = \min\left\{1, \frac{\tilde{p}(x')q\left(x^{(t)} \mid x'\right)}{\tilde{p}(x^{(t)})q\left(x' \mid x^{(t)}\right)}\right\}$$

  3. If accepted, set $x^{(t+1)} = x'$, otherwise $x^{(t+1)} = x^{(t)}$

Theorem 7.2. *The Metropolis-Hastings algorithm defines a Markov chain with stationary distribution $\pi(x)$ equal to the target distribution $p(x)$*

Proof. Notice that

$$a(x' \mid x) = \min\left\{1, \frac{\tilde{p}(x')q\left(x \mid x'\right)}{\tilde{p}(x)q\left(x' \mid x\right)}\right\} = \min\left\{1, \frac{p(x')q\left(x \mid x'\right)}{p(x)q\left(x' \mid x\right)}\right\}$$

The resulting Markov chain has the following transition probabilities

$$r(x' \mid x) = \begin{cases} q(x' \mid x)a(x' \mid x), & \text{if } x' \neq x \\ q(x \mid x) + \sum\limits_{x' \neq x} q(x' \mid x)(1 - a(x' \mid x)), & \text{if } x' = x \end{cases}$$

It suffices to show detailed balance for $x \neq x'$, i.e.

$$r(x' \mid x)p(x) = r(x \mid x')p(x')$$

Rewriting the LHS and RHS:

$$r(x' \mid x)p(x) = p(x)q(x' \mid x)\min\left\{1, \frac{p(x')q\left(x \mid x'\right)}{p(x)q\left(x' \mid x\right)}\right\} = \min\left\{p(x)q(x' \mid x), p(x')q\left(x \mid x'\right)\right\}$$

$$r(x \mid x')p(x') = p(x')q(x \mid x')\min\left\{1, \frac{p(x)q\left(x' \mid x\right)}{p(x')q\left(x \mid x'\right)}\right\} = \min\left\{p(x')q(x \mid x'), p(x)q\left(x' \mid x\right)\right\}$$

Therefore p is a stationary distribution of this Markov chain.

□

Gibbs Sampling Procedure

- Suppose the vector x has been divided into d components

$$x = (x_1, \ldots, x_d)$$

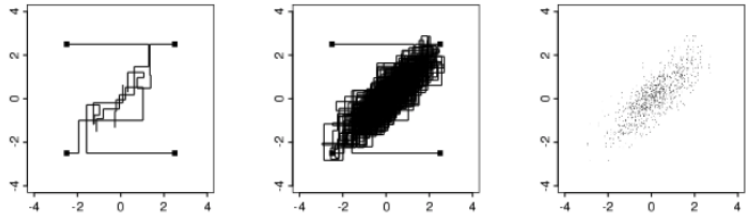
- Start with any $x^{(0)} = \left(x_1^{(0)}, \ldots, x_d^{(0)}\right)$. In the tth iteration:
  - For $j = 1, \ldots, d$, sample $x_j^{(t)}$ from the conditional distribution given other components

$$x_j^{(t)} \sim p\left(x_j \mid x_{-j}^{(t-1)}\right)$$

    where $x_{-j}^{(t-1)}$ represents all the components of x except for x_j at their current values:

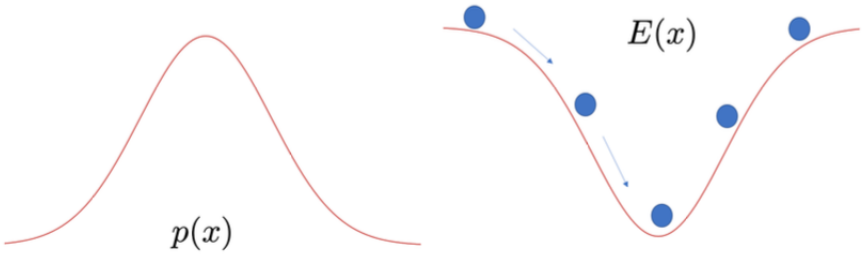
$$x_{-j}^{(t-1)} = \left(x_1^{(t)}, x_2^{(t)}, \ldots, x_{j-1}^{(t)}, x_{j+1}^{(t-1)}, \ldots, x_d^{(t-1)}\right)$$

  - No accept/reject, only accept
- Good for situations when the distribution is hard but full-conditionals are simple (e.g. Ising model)
- Visualization of Gibbs sampling from a 2D Gaussian



Hamiltonian Monte Carlo (HMC)

- Essentially a Metropolis-Hastings algorithm with a specialized proposal mechanism
- Algorithm uses a physical anlogy to make proposals
- Given the position x, the potential energy is $E(x)$



- Construct a distribution

$$p(x) \propto e^{-E(x)} \qquad \text{with} \qquad E(x) = -\log \tilde{p}(x)$$

  where $\tilde{p}(x)$ is the unnormalized density we can evaluate
- Introduce **momentum** v carrying the kinetic energy

$$K(v) = \frac{1}{2}\|v\|^2 = \frac{1}{2}v^\top v$$

- Total energy or **Hamiltonian** is

$$H(x, v) = E(x) + K(v)$$

- Energy is preserved
  - Analogous to a frictionless ball rolling from $(x, v) \to (x', v')$, where $H(x, v) = H(x', v')$
- Ideal Hamiltonian dynamics are reversible, where we can reverse v and the ball will return to its start point, i.e. $(x', -v') \to (x, -v)$
- The joint distribution is

$$p(x, v) \propto e^{-E(x)}e^{-K(v)} = e^{-H(x,v)}$$

  - Momentum is Gaussian, and independent of the position
- MCMC procedure:
  - Sample the momentum from the Gaussian

  - Simulate Hamiltonian dynamics, flip sign on the momentum
    - Hamiltonian dynamics is reversible
    - Energy is constant, i.e. $p(x, v) = p(x', v') = p(x', -v')$

- To simulate Hamiltonian dynamics, take

$$\frac{dx}{dt} = \frac{\partial H}{\partial v} = \frac{\partial K}{\partial v}$$
$$\frac{dv}{dt} = -\frac{\partial H}{\partial x} = -\frac{\partial E}{\partial x}$$

  - Notice that $\frac{dH}{dt} = 0$

Leap-Frog Integrator

- A numerical approximation that is more accurate than Euler's method

$$v\left(t + \frac{\epsilon}{2}\right) = v(t) - \frac{\epsilon}{2}\frac{\partial E}{\partial x}(x(t))$$
$$x\left(t + \epsilon\right) = x(t) + \epsilon\frac{\partial K}{\partial v}\left(v\left(t + \frac{\epsilon}{2}\right)\right)$$
$$v(t + \epsilon) = v\left(t + \frac{\epsilon}{2}\right) - \frac{\epsilon}{2}\frac{\partial E}{\partial x}\left(x\left(t + \epsilon\right)\right)$$

- ϵ is the step size hyperparameter
- We do a fixed number of leap-frog steps
- Dynamics are still deterministic (and reversible)
- Acceptance probability:

$$\min\left\{1, \frac{\exp(H(x, v))}{\exp(H(x', v'))}\right\}$$

HMC Algorithm

1. Current position: x
2. Sample momentum: $v \sim N(0, \Sigma)$
3. Run Leapfrog integrator for L steps and reach (x', v')
4. Accept new state $(x', -v')$ (or just the position x') with probability

$$\min\left\{1, \frac{\exp(H(x, v))}{exp(H(x', v'))}\right\}$$

- Low energy points are favoured

MCMC Inference

- Sample from unnormalized posterior
- Estimate statistics from simulated values of x (e.g. mean, median, quantiles, etc.)
- **Posterior predictive distribution** of unobserved outcomes can be obtained by further simulation conditional on drawn values of x

MCMC Limitations

- We don't know whether we have ran the algorithm long enough
- We could have started very far from where our distribution is
- Since there is correlation between each item of the chain (autocorrelation), we don't know what's the "effective" number of samples

Good Ideas For MCMC

- Parallel compuation: we can run multiple chains in parallel starting at different points
- We should discard some initial samples (**burn-in phase**)
- We should examine how well the chain is "mixed"

R Hat

- Start with m chains, each of length n, i.e. $X \in \mathbb{R}^{n \times m}$
  - This should be after the burn-in phase
- The **between sequence variance** is

$$B = \frac{n}{m-1} \sum_{j=1}^{m} (\overline{x}_{\cdot j} - \overline{x}_{\cdot\cdot})^2$$

  where

$$\overline{x}_{\cdot j} = \frac{1}{n} \sum_{i=1}^{n} x_{ij} \qquad \text{(individual chain mean)}$$

$$\overline{x}_{\cdot\cdot} = \frac{1}{m} \sum_{j=1}^{m} \overline{x}_{\cdot j} = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} \qquad \text{(total mean)}$$

- The **within sequence variance** is

$$W = \frac{1}{m} \sum_{j=1}^{m} s_j^2$$

  where

$$s_j^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_{ij} - \overline{x}_{\cdot j})^2$$

- If one or more chain has not mixed well, the variance of all the chains combined together should be higher than that of individual chains
- The average variance is

$$\widehat{\mathrm{Var}}^{+}(x) = \frac{n-1}{n} W + \frac{1}{n} B$$

- Define the **R-hat** coefficient as

$$\hat{R} = \sqrt{\frac{\widehat{\mathrm{Var}}^{+}(x)}{W}}$$

- If chains have not mixed well, then $\hat{R} > 1$
- **Split-$\hat{R}$**: split each chain into the first and second halves, which detects non-stationarity within a single chain

- The R-hat above does *not* work well with infinite variance – there is a newer definition of R-hat that works better in this case

Effective Sample Size

- If $x_1, \ldots, x_n$ are i.i.d. with variance σ^2, then $\text{Var}(\overline{x}_n) = \sigma^2/n$
- In general, without assuming independence

$$\text{Var}(\overline{x}) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \text{Cov}(x_i, x_j) = \frac{\sigma^2}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \text{corr}(x_i, x_j)$$

  so

$$\frac{n^2}{\sum_{i=1}^{n} \sum_{j=1}^{n} \text{corr}(x_i, x_j)}$$

  measures "effective sample size"

- Define the **effective sample size** to be

$$n_{\text{eff}} = \frac{mn}{1 + 2 \sum_{t=1}^{\infty} \rho_t}$$

  where $\rho_t = \text{corr}(x_0, x_t)$ are unknown, so we also estimate them

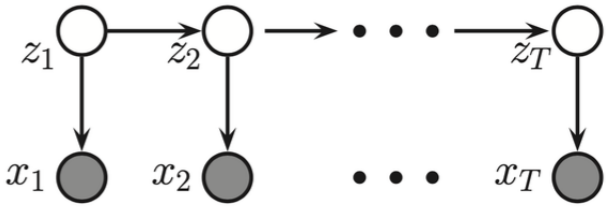
Simulation Diagnostics

- Once $\hat{R}$ is near 1 and $\hat{n}_{\text{eff}}$ is > 10 per chain *for all scalar estimands*, we collect the mn simulations (excluding the burn-in)
- Even if the iterative simulations appear to have converged, passed all tests, etc. It may still be far from convergence (i.e. all chains are stationary and well mixed)
- None of the above checks are statistically significant (i.e. no p-values)

8 Hidden Markov Models

Hidden Markov Models (HMM)

- In certain cases, Markov chain assumption is also restrictive, since the state of the variables are fully observed
- HMMs hide the temporal dependence by keeping it in the unobserved state
- No assumptions on the temporal dependence of observations is made
- For each observation x_t, we associate a corresponding unobserved hidden/latent variable z_t



- The joint distribution of the model is

$$\Pr(x_{1:T}, z_{1:T}) = \Pr(z_1) \prod_{t=2}^{T} \Pr(z_t \mid z_{t-1}) \prod_{t=1}^{T} \Pr(x_t \mid z_t)$$

- The observations are not limited by a Markov assumption of any order
- Assuming we have a homogeneous model, we only have to know three sets of distributions

1. **Initial distribution**
$$\pi(i) = \Pr(z_1 = i)$$
The probability of the first hidden variable being in state i
2. **Transition distribution**
$$\Psi(i, j) = \Pr(z_{t+1} = j \mid z_t = i)$$
The probability of moving from hidden state i to hidden state j
3. **Emission probability**
$$\psi_t(i) = \Pr(x_t \mid z_t = i)$$
The probability of an observed random variable x given the state of the hidden variable that "emitted" it

HMM Objectives

1. Compute the probability of a latent sequence given an observation sequence, i.e. compute $\Pr(z_{1:t} \mid x_{1:t})$
   - Achieved with the **forward-backward algorithm**
2. Infer the most likely sequence of hidden states, i.e. compute
$$z^* = \arg\max_{z_{1:T}} \Pr(z_{1:T} \mid x_{1:T})$$
   - Achieved using the **Viterbi algorithm**

Forward Algorithm

- Goal is to recursively compute the filtered marginals

$$\alpha_t(j) = \Pr(z_t = j \mid x_{1:t})$$

in an HMM

- Assuming that we know the initial $\Pr(z_1)$, transition $\Pr(z_t \mid z_{t-1})$, and emission $\Pr(x_t \mid z_t)$ probabilities for every t
- This is a step in the **forward-backward algorithm**
- Algorithm has 2 steps
  1. Prediction step: compute one-step-ahead predictive density

$$\Pr(z_t = j \mid x_{1:t-1}) = \sum_i \Pr(z_t = j \mid z_{t-1} = i) \Pr(z_{t-1} = i \mid x_{1:t-1}) = \sum_i \Psi(i,j)\alpha_{t-1}(i)$$

  2. Update step:

$$\begin{aligned}\alpha_t(j) &= \Pr(z_t = j \mid x_{1:t}) \\ &= \Pr(z_t = j \mid x_{1:t-1}, x_t) \\ &\propto \Pr(x_t \mid z_t = j, x_{1:t-1}) \Pr(z_t = j \mid x_{1:t-1}) \\ &\propto \Pr(x_t \mid z_t = j) \Pr(z_t = j \mid x_{1:t-1}) \\ &= \psi_t(j) \Pr(z_t = j \mid x_{1:t-1})\end{aligned}$$

  where the normalizing constant is

$$Z_t = \Pr(x_t \mid x_{1:t-1}) = \sum_j \Pr(z_t = j \mid x_{1:t-1}) \Pr(x_t \mid z_t = j)$$

- The above process is called the *predict-update cycle*
- Using matrix notation, we can write the update as

$$\alpha_t \propto \psi_t \odot \left(\Psi^\top \alpha_{t-1}\right)$$

where

  - $\psi_t(j) = \Pr(x_t \mid z_t = j)$ is the local evidence at time t
  - $\Psi(i,j) = \Pr(z_t = j \mid z_{t-1} = i)$ is the transition matrix
  - $\odot$ is the Hadamard (entrywise) product

Forward-Backward Algorithm

- The forward-backward algorithm is used to efficiently estimate the latent sequence given an observation sequence under a HMM
- Want to compute

$$\Pr(z_t \mid x_{1:T}) \qquad \forall t \in [1, T]$$

assuming that we know the initial $\Pr(z_1)$, transition $\Pr(z_t \mid z_{t-1})$, and emission $\Pr(x_t \mid z_t)$ probabilities for every t

- Tasks of hidden state inference
  1. **Filtering**: compute posterior over current hidden state, $\Pr(z_t \mid x_{1:t})$
  2. **Prediction**: compute posterior over future hidden state: $\Pr(z_{t+k} \mid x_{1:t})$

3. **Smoothing**: compute posterior over past hidden state, $\Pr(z_k \mid x_{1:t})$, where $1 < k < t$

- The probability of interest, $\Pr(z_t \mid x_{1:T})$ is computed using a forward and backward recursion
  1. **Forward recursion**: $\Pr(z_t \mid x_{1:t})$
  2. **Backward recursion**: $\Pr(x_{t+1:T} \mid z_t)$
- We can break the chain into two parts: the past and the future, by conditioning on z_t
- We have

$$\begin{aligned}\gamma_t &= \Pr(z_t \mid x_{1:T}) \\ &\propto \Pr(z_t, x_{1:T}) \\ &= \Pr(z_t, x_{1:t}) \Pr(x_{t+1:T} \mid z_t, x_{1:t}) \\ &= \Pr(z_t, x_{1:t}) \Pr(x_{t+1:T} \mid z_t) \\ &\propto (\text{Forward recursion})(\text{Backward recursion})\end{aligned}$$

Backward Recursion

- In the backward pass:

$$\begin{aligned}\beta_t(i) &= \Pr(x_{t+1:T} \mid z_t = i) \\ &= \sum_j \Pr(z_{t+1} = j, x_{t+1:T} \mid z_t = i) \\ &= \sum_j \Pr(x_{t+2:T} \mid z_{t+1} = j, z_t = i, x_{t+1}) \Pr(x_{t+1} \mid z_{t+1} = j, z_t = i) \Pr(z_{t+1} = j \mid z_t = i) \\ &= \sum_j \Pr(x_{t+2:T} \mid z_{t+1} = j) \Pr(x_{t+1} \mid z_{t+1} = j) \Pr(z_{t+1} = j \mid z_t = i) \\ &= \sum_j \beta_{t+1}(j) \psi_{t+1}(j) \Psi(i, j)\end{aligned}$$

- In vector notation:

$$\beta_t = \Psi(\psi_{t+1} \odot \beta_{t+1})$$

  where $\beta_T(i) = 1$
- Once we have the forward and the backward steps complete, we can compute

$$\gamma_t(i) \propto \alpha_t(i) \beta_t(i)$$

  which is the **forward-backward algorithm**

Viterbi Algorithm

- The Viterbi algorithm is used to compute the most probable sequence

$$\hat{z} = \arg\max_{z_{1:T}} \Pr(z_{1:T} \mid x_{1:T})$$

- The forward pass uses the max-product, and the backward pass uses a traceback procedure to recover the most probable path
- Define

$$\delta_t(j) = \max_{z_{1:t-1}} \Pr(z_{1:t-1}, z_t = j \mid x_{1:t})$$

  which is the probability of ending up in state j at time t, by taking the most probable path

- Notice that

$$\begin{aligned}\delta_t(j) &= \max_{z_{1:t-1}} \Pr(z_{1:t-1}, z_t = j \mid x_{1:t}) \\ &\propto \max_{z_{1:t-1}} \Pr(z_{1:t-2}, z_{t-1} = i \mid x_{1:t-1}) \Pr(z_t = j \mid z_{t-1} = i) \Pr(x_t \mid z_t = j) \\ &= \max_i \delta_{t-1}(i) \Psi(i,j) \psi_t(j)\end{aligned}$$

- We keep track of the most likely previous state

$$\theta_t(j) = \arg\max_i \delta_{t-1}(i) \Psi(i,j) \psi_t(j)$$

- Initialize the algorithm with

$$\delta_1(j) = \pi_j \psi_1(j)$$

  where $\pi_j = \Pr(z_1 = j)$

- Terminate the algorithm with

$$z_T^* = \arg\max_i \delta_T(i)$$

- Then compute the most probable sequence of states using traceback

$$z_t^* = \theta_{t+1}(z_{t+1}^*)$$

9 Variational Inference

Posterior Inference For Latent Variable Models

- Latent variable models have a factorizatoin $\Pr(x, z) = \Pr(z)\Pr(x \mid z)$ where
  - x are the observations or data
  - z are the unobserved (latent) variables
  - $\Pr(z)$ is called the **prior**
  - $\Pr(x \mid z)$ is called the **likelihood**
  - The conditional distribution of the unobserved variables given the observed variables (aka the **posterior**) is
$$\Pr(z \mid x) = \frac{\Pr(x, z)}{\Pr(x)} = \frac{\Pr(x \mid z)\Pr(z)}{\int \Pr(x, z)dz}$$

Expensive Operations About the Posterior

- Computing the evidence/marginal likelihood $\Pr(x) = \int \Pr(z, x)dz$ is intractable when z is high-dimensional
- Computing the posterior $\Pr(z \mid x) = \frac{\Pr(x,z)}{\Pr(x)}$
- Computing marginals of $\Pr(z_1 \mid x) = \int \Pr(z_1, \ldots, z_D \mid x)dz_2dz_3 \cdots dz_D$ (e.g. finding the posterior over a single player's skill given all games)
- Sampling $z \sim \Pr(z \mid x)$

Variational Methods

- Variatiional inference is closely related to the calculus of variations
- Calculus of variations is the calculus of functionals (which take functions as arguments)
- Variational inference is an approximate inference method where we seek a tractable (e.g. factorized) approximation to the target intractable distribution
- Formally, variational inference works as follows:
  - Choose a tractable distribution $q(z) \in Q$ from a feasible set Q; this distribution will be used to approximate $p(z \mid x)$
  - Encode some notion of "difference" between $p(z \mid x)$ and q that can be efficiently estimated (usually using KL divergence)
  - Minimize this difference (usually through an iterative optimization method)
- Whatever feasible set we choose for Q, it's usually not the case that there is any $q \in Q$ that exactly matches the true posterior

KL Divergence

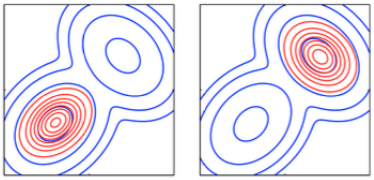
- We measure the difference between q and p using the **Kullback-Leibler divergence**
$$KL(q(z) \,||\, p(z \mid x)) = \int q(z) \log \frac{q(z)}{p(z \mid x)} dz = \mathbb{E}_{z\sim q} \log \frac{q(z)}{p(z \mid x)}$$
- Properties of KL divergence
  - $KL(q \,||\, p) \geq 0$

  - $KL(q \,||\, p) = 0 \iff q = p$
  - $KL(q \,||\, p) \neq KL(p \,||\, q)$
    - Therefore KL divergence is not a metric, since it's not symmetric
- We could minimize either $KL(q \,||\, p)$ or $KL(p \,||\, q)$, and we will choose the tractable one

Information (I-) Projetion

$$q^* = \arg\min_{q \in Q} KL(q \,||\, p) = \arg\min_{q \in Q} \mathbb{E}_{x \sim q(x)} \log \frac{q(x)}{p(x)}$$

- $p \approx q$ means $KL(q \,||\, p)$ is small
- I-projection underestimates support, and does not yield the correct moments
- $KL(q \,||\, p)$ penalizes q having mass where p does not

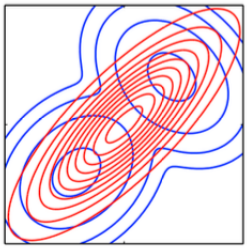


p=Blue, q^*=Red (two equivalently good solutions!)

Moment (M-) Projection

$$q^* = \arg\min_{q \in Q} KL(p \,||\, q) = \arg\min_{q \in Q} \mathbb{E}_{x \sim p(x)} \log \frac{p(x)}{q(x)}$$

- $p \approx q$ means $KL(p \,||\, q)$ is small
- $KL(p \,||\, q)$ penalizes q missing mass where p has some
- M-projection yields a distribution $q(x)$ with the correct mean and covariance



p=Blue, q^*=Red

Maximum Entropy Interpretation

- **Entropy** meausures the uncertainty in the distribution p:

$$H(p) = -\mathbb{E}_{x \sim p(x)} \log p(x)$$

- Consider the optimization problem:
  - Maximize $H(p)$
  - Subject to $\mathbb{E}_{x \sim p(x)}[f_i(x)] = t_i$ for $i = 1, \ldots, k$, where t_i are constants
- Exponential family of distributions maximize the entropy $H(p)$ over all distributions satisfying the constraints

- In M-projection, if Q is a set of exponential families, then the expeccted sufficient statistics w.r.t. $q^*(x)$ is the same as that w.r.t. $p(x)$
- M-projection require expectation w.r.t. p, hence intractable
- Most variational inference algorithms make use of the I-projection

Mean-Field Approach

- Given an arbitrary MRF
$$p(x \mid \theta) = \exp\left\{\sum_{c\in\mathcal{C}} \phi_c(x_c) - \log Z(\theta)\right\}$$
- We find an approximate distribution $q(x) \in Q$ by performing I-projection to $p(x)$
$$\begin{aligned} q^* &= \arg\min_{q\in Q} KL(q \,||\, p) \\ &= \arg\min_{q\in Q} \mathbb{E}_{x\sim q(x)} \log \frac{q(x)}{p(x \mid \theta)} \\ &= \arg\min_{q\in Q} \mathbb{E}_{x\sim q(x)} \left[\log q(x) - \sum_{c\in\mathcal{C}} \phi_c(x_c) + \log Z(\theta)\right] \\ &= \arg\max_{q\in Q} \sum_{c\in\mathcal{C}} \mathbb{E}_q[\phi_c(x_c)] + H(q) \end{aligned}$$
- For tractability, we need a nice set Q
- If $p \in Q$, then $q^* = p$, but this almost never happens

Naive Mean-Field

- We can use the mean-field approach and assume
$$q(x) = \prod_{i\in V} q_i(x_i)$$
where the set Q is composed of those distributions that factor out
- Using this in the maximization problem, we can simplify to
$$q^* = \arg\max_{q\in Q} \sum_{c\in\mathcal{C}} \sum_{x_c} q(x_c)\phi_c(x_c) + H(q)$$
- It can be shown that
$$q(x_c) = \prod_{i\in c} q_i(x_i) \qquad \text{and} \qquad H(q) = \sum_i H(q_i)$$
- Therefore
$$q^* = \arg\max_q \sum_{c\in\mathcal{C}} \sum_{x_c} \phi_c(x_c) \prod_{i\in c} q_i(x_i) + \sum_i H(q_i)$$
subject to
$$q_i(x_i) \geq 0 \qquad \text{and} \qquad \sum_{x_i} q_i(x_i) = 1$$

- If we assume that we have a pairwise MRF, we can further simplify to

$$q^* = \arg\max_q \sum_{(i,j)\in E} \sum_{x_i, x_j} \phi_{ij}(x_i, x_j) q_i(x_i) q_j(x_j) - \sum_i \sum_{x_i} q_i(x_i) \log(q_i(x_i))$$

  subject to

$$q_i(x_i) \geq 0 \qquad \text{and} \qquad \sum_{x_i} q_i(x_i) = 1$$

Coordinate Maximization

- Difficult since we may have many local maxima
- Can try to optimize using block coordinate ascent:
  - Initialize $\{q_i(x_i)\}_{i\in V}$ uniformly
  - Iterate over $i \in V$:
    - Greedily maximize the objective over $q_i(x_i)$
    - Equivalent to

$$q_i(x_i) \propto \exp\left\{ \sum_{j\in N(i)} \sum_{x_j} q_j(x_j) \phi_{ij}(x_i, x_j) \right\}$$

    - Which follows from: write the Lagrangian, take the derivative, set to zero, and solve for $q_i(x_i)$
    - Repeat until convergence
- Guaranteed to converge, but can converge to local optima

Evidence Lower Bound

- Choose a tractable parametric distribution $q_\phi(z)$ with parameters ϕ, and use it to approximate $p(z \mid x)$
- Evaluating $KL(q_\phi(z) \,||\, p(z \mid x))$ is intractable because of the integral over z and the term $p(z \mid x)$

$$KL(q_\phi(z) \,||\, p(z \mid x)) = \int q_\phi(z) \log \frac{q_\phi(z)}{p(z \mid x)} dz$$

- Can optimize this KL without knowing the normalization constant $p(x)$
- Can solve a surrogate optimization problem: maximize the **evidence lower bound (ELBO)**
  - Equivalent to minimizing the KL divergence

$$\begin{aligned} KL(q_\phi(z) \,||\, p(z \mid x)) &= \mathbb{E}_{z\sim q_\phi} \log \frac{q_\phi(z)}{p(z \mid x)} \\ &= \mathbb{E}_{z\sim q_\phi} \log \left(q_\phi(z) \cdot \frac{p(x)}{p(z, x)} \right) \\ &= \mathbb{E}_{z\sim q_\phi} \log \frac{q_\phi(z)}{p(z, x)} + \mathbb{E}_{z\sim q_\phi} \log p(x) \\ &= -\mathcal{L}(\phi) + \log p(x) \end{aligned}$$

  where $\mathcal{L}(\phi)$ is the ELBO

$$\mathcal{L}(\phi) = \mathbb{E}_{z\sim q_\phi} \left(\log p(z, x) - \log q_\phi(z) \right)$$

- Can optimize using gradient descent, need to compute an unbiased estimate of $\nabla_\phi \mathcal{L}(\phi)$

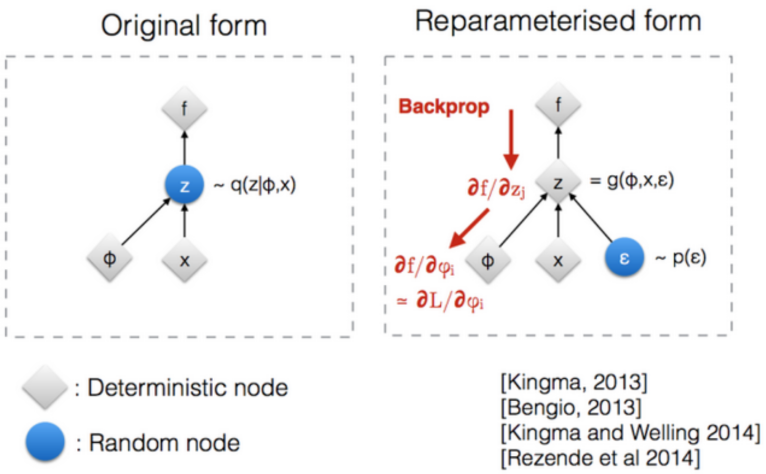
Reparameterization Trick

- Need to sample $z \sim q_\phi(z)$ to estimate the gradient of ELBO with simple Monte Carlo
- The expectation is over $q_\phi(z)$, which depends on ϕ

$$\mathcal{L}(\phi) = \mathbb{E}_{z \sim q_\phi} \left(\log p(x, z) - \log q_\phi(z) \right)$$

- Can break the sampling process into 2 parts
  1. Sample a random variable ϵ that has fixed (or no) parameters (e.g. uniform distribution, standard normal)
  2. Deterministically compute zs as a function of ϕ and ϵ, such that

$$\epsilon \sim p(\epsilon) \wedge z = T(\epsilon, \phi) \implies z \sim q_\phi(z)$$



- This makes the density independent of the parameter ϕ, which allows us to use simple Monte Carlo

$$\begin{aligned}
\nabla_\phi \mathcal{L}(\phi) &= \nabla_\phi \mathbb{E}_{z \sim q_\phi(z)} \left(\log p(x, z) - \log q_\phi(z) \right) \\
&= \nabla_\phi \mathbb{E}_{\epsilon \sim p(\epsilon)} \left(\log p(x, T(\phi, \epsilon)) - \log q_\phi(T(\phi, \epsilon)) \right) \\
&= \mathbb{E}_{\epsilon \sim p(\epsilon)} \nabla_\phi \left(\log p(x, T(\phi, \epsilon)) - \log q_\phi(T(\phi, \epsilon)) \right)
\end{aligned}$$

Score Function Gradient Estimator

$$\nabla_\phi \mathbb{E}_{z \sim q_\phi(z)} f(z) = \nabla_\phi \int f(z) q_\phi(z) dz$$

- Using this, it can be shown that

$$\nabla_\phi \mathcal{L}(\phi) = \mathbb{E}_{z \sim q_\phi(z)} \left[\log p_\theta(x \mid z) \nabla_\phi \left(\log q_\phi(z) \right) \right] - \nabla_\phi KL(q_\phi \mid p)$$

  which is unbiased, but has high variance

SVI: Stochastic Variational Inference

- We can compute a simple Monte Carlo estimate of the gradient
- Can work with a minibatch of size m:

$$\begin{aligned}
&\hat{\mathbb{E}}_{\epsilon \sim p(\epsilon)} \nabla_\phi \left[\log p(x, T(\phi, \epsilon)) - \log q_\phi(T(\phi, \epsilon)) \right] \\
&\approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\phi \left[\log p(x, T(\phi, \epsilon_i)) - \log q_\phi(T(\phi, \epsilon_i)) \right]
\end{aligned}$$

MCMC vs. SVI

MCMC	**SVI**
Asymptotically exact	Can get stuck at a bad approximate distribution
Theoretically guaranteed	Little guarantees
Well-understood	Limited flexibility of variational approximation
Hard to assess convergence	Can tell if making progress
Hard to tune hyperparameters	Simple
Can't easily use minibatches	Can use minibatches for fitting to large datasets

10 Variational Autoencoders

Nonlinear Dimension Reduction

- Achieved by having the same number of outputs as inputs
- Consider a MLP with D inputs, D outputs, and M hidden units where $M < D$
  - Called an *autoencoder*
- Can squeeze the information through some kind of bottleneck
- If a linear network (i.e. linear activation) is used, then the autoencoder is equivalent to PCA

Autoencoders and PCA

- Given an input $\mathbf{x}$, its corresponding reconstruction is given by

$$y_k(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{M} w_{kj}^{(2)} \sigma \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i \right), \quad k = 1, \ldots, D$$

- Can determine the network parameters $\mathbf{w}$ by minimizing the reconstruction error:

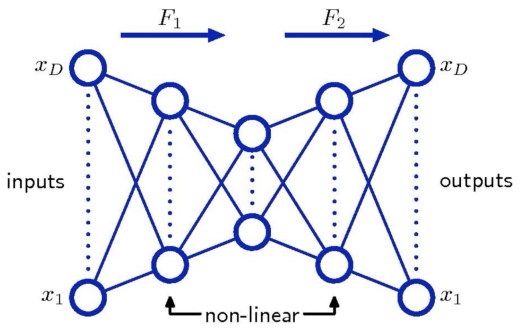
$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \| y(\mathbf{x}_n, \mathbf{w}) - \mathbf{x}_n \|^2$$

Autoencoders

- Encoder: $g(x) = z \in F, x \in X$
- Decoder: $f(z) = \tilde{x} \in X$
- X is the data space and F is the feature (latent) space
- z is the code, compressed representation of the input x
- The code must be a bottlenet, i.e. $\dim F \ll \dim X$
- Goal: $\tilde{x} = f(g(x)) \approx x$

Deep Autoencoders

- Can put extra nonlinear hidden layers between the input and the bottlenect and between the bottleneck and output



- Can be trained by the minimization of the reconstruction error function

- Hard to train
- Can produce better reconstructions than PCA

Issues with Autoencoders

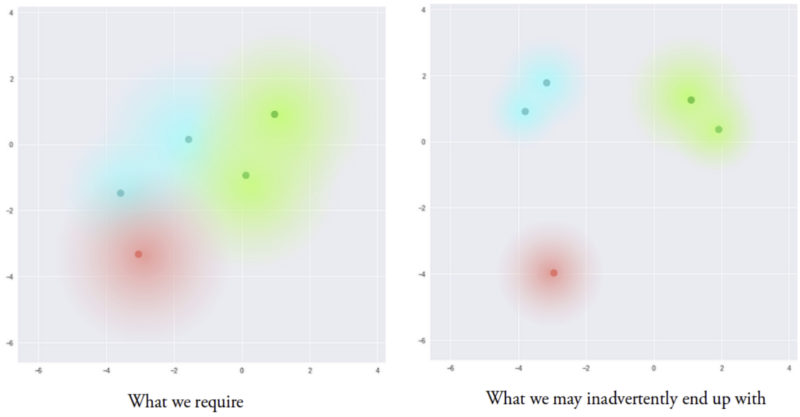
1. Proximity in data space does not mean proximity in feature space
   - The codes learned by the model are deterministic, i.e.

$$g(x_1) = z_1 \implies f(z_1) = \tilde{x}_1 \qquad g(x_2) = z_2 \implies f(z_2) = \tilde{x}_2$$

   - However

$$x_1 \approx x_2 \not\Rightarrow z_1 \approx z_2$$

   - The latent space may not be continuous, or allow easy interpolation



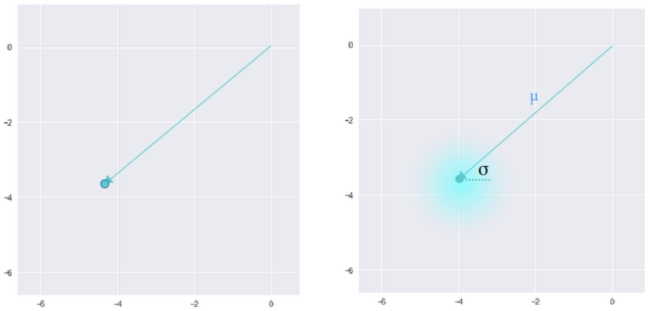
What we require

What we may inadvertently end up with

2. We cannot accurately measure the goodness of a reconstruction
   - Choosing a distance metric may result in penalization of translation in reconstructions
   - Generally hard to choose an appropriate metric

Variational Autoencoders (VAEs)

- Encode inputs with uncertainty
- The encoder of a VAE outputs a probability distribution $q_\phi(z \mid x)$
- The encoder learns two vectors, means μ and standard deviations σ
  - μ controls where encoding of input is centered
  - σ controls how much can the encoding vary



Standard Autoencoder
(direct encoding coordinates)

Variational Autoencoder
(μ and σ initialize a probability distribution)

- Encodings are generated at random from the "circle", and the deccoder learns that all nearby points refer to the same input
- The model is generated by the joint distribution over the latent codes and the input data $p(x, z)$

$$p(x, z) = \text{prior} \times \text{likelihood} = p(z)p(x \mid z)$$

- The encoder is $p(z \mid x) = p(x, z)/p(x)$ (probability of code given data)
- However, learning $p(x) = \int p(x \mid z)p(z)dz$ is intractable
- We can introduce an approximation with its own set of parameters, q_ϕ, and learn these parameters such that

$$q_\phi(z \mid x) \approx p(z \mid x)$$

- An idea from variational inference: maximize ELBO

$$\mathcal{L}(\theta, \phi; x) = \text{ELBO} = \mathbb{E}_{z \sim q_\phi}\left[\log p_\theta(x \mid z)\right] - KL(q_\phi(z \mid x) \mid\mid p(z))$$

  - First term is expected log-likelihood
  - Second term is the divergence of q_ϕ from the true prior
- Use this as loss function when training VAEs
- The VAE structure becomes
  - Encoder: $q_{\phi_i}(z \mid x_i) = \mathcal{N}(\mu_i, \sigma_i^2)$ where $\phi_i = (\mu_i, \log \sigma_i)$
  - Decoder: $f(z_i) = \theta_i$, which is typically a neural network

VAE Pipeline

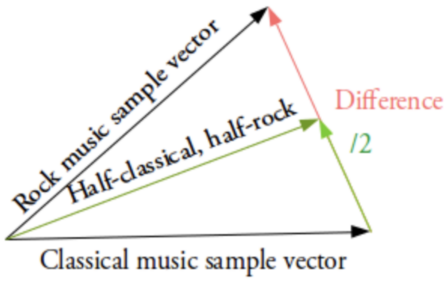
- For a given input (or minibatch) x_i:
  1. Sample $z_i \sim q_{\phi_i}(z \mid x_i)$ as the code in the feature space
  2. Run the code through decoder and write the likelihood $p_\theta(x \mid z)$
  3. Compute the loss function

$$\mathcal{L}(x; \theta, \phi) = -\mathbb{E}_{z_\phi \sim q_\phi}\left[\log p_\theta(x \mid z)\right] + KL(q_\phi(z \mid x) \mid\mid p(z))$$

- Use gradient-based optimization to backpropogate $\nabla_\theta L$ and $\nabla \phi L$
- After VAE is trained, we can sample new inputs

$$z \sim p(z) \qquad \tilde{x} \sim p_\theta(x \mid z)$$

- Can interpolate between inputs, using simple vector arithmetic



The Reparametrization Trick

- Encoder generates a code by sampling from the distribution $q_\phi(z \mid x)$
- With this sampling process, gradients are blocked from flowing into the encoder
- Use the reparametrization trick to solve this problem: instead of sampling z directly from its distribution (e.g. $z_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$), we can express z_i as

$$z_i = \mu_i + \sigma_i \times \epsilon_i \qquad \text{where} \epsilon_i \sim \mathcal{N}(0, I)$$

  with this, gradients can flow through the entire network

Amortized Inference

- Instead of doing VI from scratch every time we see a new datapoint, we can learn a function that looks at the data x_i and output an approximate posterior $q_\phi(z_i \mid x_i)$
  - We call this a **recognition model**
- Instead of a separate ϕ_i for each data exmple, we can have a single global ϕ that specifies the parameters of the recognition model
- Can do this with a neural network
- Can have a network take in x_i, and output the mean and variance vector for a Gaussian:

$$q_\phi(z_i \mid x_i) = \mathcal{N}(z_i \mid \mu_\phi(x_i), \sigma_\phi(x_i f))$$

VAE vs. Amortized VAE

- For a given input (or minibatch) x_i:
  - Standard VAE samples

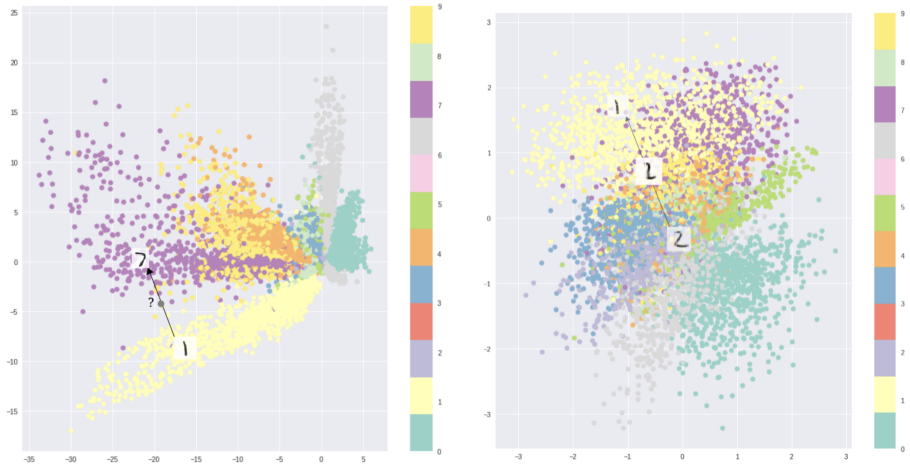
$$z_i \sim q_{\phi_i}(z \mid x_i) = \mathcal{N}(\mu_i, \sigma_i^2 I)$$

  - Amortized VAE samples

$$z_i \sim q_\phi(z \mid x_i) = \mathcal{N}(\mu_\phi(x_i), \sigma_\phi(x_i))$$

- The rest of the pipeline for VAE and Amortized VAE are the same
- Amortized VAE allows using the shared parameters for all data points, which reduces the number of parameters for the encoder
- Standard VAE encoder is more expressive

Autoencoder vs. VAE



VAE Loss Interpretation

- The VAE maximization objective can be written as

$$\mathcal{L}(x;\theta,\phi) = \mathbb{E}_{z_\phi \sim q_\phi}\left[\log p_\theta(x \mid z)\right] - KL(q_\phi(z \mid x) \,||\, p(z)) \tag{1}$$
$$= \mathbb{E}_{z_\phi \sim q_\phi}\left[\log p_\theta(x,z)\right] + H(q_\phi) \tag{2}$$

- (1): maximize expected log-likelihood while penalizing solutinos that are different from the prior
- (2): maximize expected complete data log-likelihood while penalizing low entropy solutions

VAE vs. EM

- In addition to the above, VAE performs alternating gradient descent/ascent
- Expectation in EM algorithm maximizes

$$\mathcal{Q}\left(\phi, \phi^{\text{old}}\right) = \mathbb{E}_{z \sim q_\phi^{\text{old}}}\left[\log p_\phi(x,z)\right]$$

  - This maximizes expected copmlete data log-likelihood while the expectation is over the posterior
  - We perform maximization at each iteration

VAE in Image Denoising

- We can train a *denoising autoencoder* to perform image denoising
- We can feed the *noisy* image into the encoder, then minimize the reconstruction error between the decoder output and the *original* image
- This method requires training and knowledge of the noise structure (i.e. fully supervised)
- In contrast, loopy BP works for a single noisy image and does not require the knowledge of noise structure (i.e. unsupervised)

11 Embeddings

NLP Concepts

- **Token**: a single "atom" of text, usually a word
- **Documents**: a complete datapoint of text
- **Span**: a contiguous sequence of tokens
- **Vocabulary**: a list of all possible tokens

NLP Tasks

- Classification (documents, spans, tokens)
  - Hate speech detection
  - Spam filtering
  - Social media drug adverse effect identification
- Generation (question answering, summarization, free-text generation)
  - Translating natural language into SQL queries
  - Voice assistant
  - Chatbots
- Regression (essay scoring, like count prediction)
  - E.g. how many retweets will this tweet get?
- Information extraction
  - E.g. who are the people mentioned in this text?
- Document retrieval
  - Google search
  - Automatic literature review

Example: Spam

- Consider a set of observations $(x_i, y_i)_{i=1:N}$ where $y_i = 1$ means datapoint i was spam, and x_i is the text of the email
- Goal: create/learn a function f_θ such that $f_\theta(x_i) = \Pr(y_i \mid x_i)$

Heuristics

- If x_i contains any prescription medication name, then $\hat{y}_i = 1$
- If x_i is mostly capital letters, then $\hat{y}_i = 1$
- If x_i contains "Make {amount} every {time period}", then $\hat{y}_i = 1$

Bag of Words

- To apply learning algorithms, we need to convert x into a numerical representation h
- **(Binary) Bag of Words**: given a vocabulary $V_{j=1:M}$, determine h such that $h_j = 1$ whenever token j from the vocabulary is present in x

- Each datapoint x is represented by an M-dimensional vector of 0s and 1s
- To determine the vocabulary, we can list and count all the words in all the documents, then keep the top M most frequent words
- Having numerical features, we can plug them as input into any "standard" algorithm
- Only cares about the presence/absence of each word, can extend to counting the number of times each word appears
- Does not account for phrases (where only that specific combination of word makes sense)

N-Grams

- An **N-gram** is a contiguous sequencec of N tokens from a given text
- Under $N = 1$, called *unigrams*, for $V = (\text{This}, \text{is}, \text{a}, \text{sentence})$:

$$\text{"This is a sentence"} = (1, 1, 1, 1)$$
$$\text{"A sentence"} = (0, 0, 1, 1)$$

- Under $N = 2$, called *bigrams*, for $V = (\text{This is}, \text{is a}, \text{a sentence})$:

$$\text{"This is a sentence"} = (1, 1, 1)$$
$$\text{"A sentence"} = (0, 0, 1)$$

Common Words

- Some common words do not contribute in terms of distinguishing between texts
  - E.g. words like "the", "a", etc.
  - In practice, we often remove them from all text and completely ignore them
    - Those words are called **stopwords**
- We want to *learn* which words to include in the vocabulary
- We can better represent documents by the **relative frequency** of words in them

Term Frequency

- We can represent the **term frequency**, i.e. count, of a word in the following ways:
  - Raw count of times it is present in x: **BoW**
  - Binarized count of times it is present in x: **binary BoW**
  - Count of times it is present in x divided by number of tokens in x
  - Raw count scaled by number of other terms in x
  - Log of raw count

Inverse Document Frequency

- Denote

$$N_j = \sum_{i=1}^{N} \mathbb{1}\left(V_j \in x_j\right)$$

  count of datapoints that include the jth word in the vocabulary

- We can represent the **inverse document frequency**, i.e. relative prevalence or how common a word is, in the following ways:
  - $1/N_j$
  - N/N_j
  - $\log(N/N_j)$
- Rare words have higher IDF since rare words are likely to be more significant in distinguishing between texts

TF-IDF

- By combining term frequency with inverse document frequency, we can measure how common the word is in a particular datapoint relative to other documents

$$\text{TF-IDF}(x) = \text{TF}(x) \times \text{IDF}(x)$$

Embeddings

- The above methods are used to generate numerical representations for whole documents, by the use of word occurrences and simple functions
- h_i is the **embedding** of x_i
- $g(x) = h$ is an embedding function
- We can reframe our problem of learning $f_\theta(x) = y$ as

$$f_\theta(x) = c_{\theta_1}(h) = c_{\theta_1}\left(g_{\theta_2}(x)\right) = y$$

  where c_θ is any classification/regression function, and g_θ is an embedding function
- Embeddings are not restricted to documents, we could also embed an image

Word2Vec

- Assumption: words that have similar meanings will occur in similar contexts
- Define a **context** of size k of token $w_{i,j}$ as a set of tokens (also call a **skip-gram**)

$$\text{context}(w_{i,j}) = \left\{w_{i,j-k}, w_{i,j-(k-1)}, \ldots, w_{i,j-1}, w_{i,j+1}, \ldots, w_{i,j+k}\right\}$$

- Given a set of datapoints $x_{i=1:N}$ consisting of R_i words each, and a vocabulary $V_{r=1:M}$, we define an unsupervised learning task of predicting what words occur in the context of each word in the vocabulary
- Formally, given a sequence of training words $w_1, \ldots, w_T$, we want to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j\in\text{context}(t)}\log\Pr(w_j \mid w_t)$$

- Can formulate $\Pr(x \mid w)$ using the softmax function:

$$\Pr(x \mid w) = \frac{\exp\left(u(w)^\top v(x)\right)}{\sum_{m=1}^{M}\exp\left(u(w)^\top v(V_m)\right)}$$

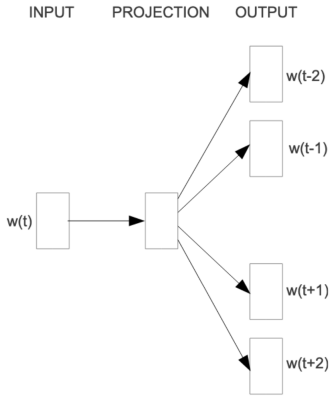
  where $u(w)$ is the "word" and $v(w)$ is the context representation of the word w

– Can take u and v to be simple lienar projections of the *one-hot* (binary BoW) encoding of the word and context, respectively, i.e.

$$u(w) = bBoW(w)U^\top$$

– The matrix U is of size $R \times e$ where e is the embedding dimension (hyperparameter)

– Can define v similarly as a linear projection, introduce matrix V analogous to U



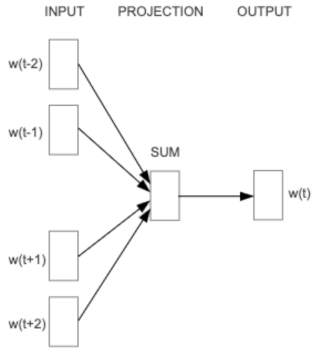
Skip-gram

- Training process

  1. Initialize U and V
  2. Sample pairs of words (w_i, w_j) and compute the objective
  3. Gradient descent based on the objective
  4. After convergence, keep only the matrix U
  5. Embedding of word at vocabulary index i is the ith row of U

Continuous Bag of Words (CBoW)

- Objective for predict the word from its context



CBOW

Properties of Word2Vec

- Let $h(x)$ be the Word2Vec embedding of the word x, then we can perform vector algebra, e.g.

$$h(\text{“Athens”}) - h(\text{“Greece”}) + h(\text{“Germany”}) \approx h(\text{“Berlin”})$$
$$h(\text{“Mice”}) - h(\text{“Mouse”}) + h(\text{“Dollar”}) \approx h(\text{“Dollars”})$$

- However the training procedure is infeasible, since we would have to compute softmax over the entire vocabulary at every step

Negative Sampling

- Instead of evaluating the softmax with a target of 1 for one dimension and 0 for all others, we can only take a small sample of negative examples in the vocabulary
  - For large datasets, can only take 2-5 negative examples
  - For small datasets, can take 5-20 negative examples

$$p(w_i) = \frac{C_{w_i}^{3/4}}{Z}$$

  - $p(w_i)$ is the probability of selecting word w_i as a negative example (i.e. sampling)
  - C_{w_i} is the count of word w_i in the corpus
  - Z is a normalization constant

Token Classification and Embeddings

- Can use Word2Vec embedding of each token as an input to a classifier
- Can combine different embedding methods, e.g. TF-IDF with Word2Vec

Generative Tasks

- Can train in the unsupervised setting of CBoW, with the modified context
- Sample the next word, conditional on the previous k based on the CBoW softmax

Similarity

- Can measure similarity using the **cosine similarity**:

$$\text{cosine sim}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

12 Kernel Methods

Linear Regression

- Given a training set of inputs and targets $\left\{\left(\mathbf{x}^{(i)}, t^{(i)}\right)\right\}_{i=1}^{N}$, a linaer model is given by

$$\mathbf{y} = \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x})$$

  where $\boldsymbol{\psi}(\mathbf{x})$ is the feature map

- When vectorized, we have the design matrix $\mathbf{X}$ in the input space and

$$\boldsymbol{\Psi} = \begin{bmatrix} - & \boldsymbol{\psi}(\mathbf{x}^{(1)}) & - \\ - & \boldsymbol{\psi}(\mathbf{x}^{(2)}) & - \\ & \cdots & \\ - & \boldsymbol{\psi}(\mathbf{x}^{(N)}) & - \end{bmatrix}$$

  to generate predictions

$$\mathbf{y} = \boldsymbol{\Psi}\mathbf{w}$$

Linear Regression as Maximum Likelihood

- Assume a Gaussian noise model

$$t \mid \mathbf{x} \sim \mathcal{N}\left(\mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x}), \sigma^2\right)$$

  and a Gaussian prior

$$\mathbf{w} \sim \mathcal{N}(\mathbf{m}, \mathbf{S})$$

- The MLE under the first model leads to the standard least squares
- The MAP under these two leads to Ridge regression (if $\mathbf{m} = \mathbf{0}$ and $\mathbf{S} = \alpha^{-1}\mathbf{I}$)

Regularized Linear Regression as MAP Estimation

- MAP inference:

$$\arg\max_{\mathbf{w}} \log \Pr(\mathbf{w} \mid \mathcal{D}) = \arg\max_{\mathbf{w}} \left[\log \Pr(\mathbf{w}) + \log \Pr(\mathcal{D} \mid \mathbf{w})\right]$$

- Log likelihood term:

$$\log \Pr(\mathcal{D} \mid \mathbf{w}) = \text{const} - \frac{1}{2\sigma^2} \sum_{i=1}^{N} \left(t^{(i)} - \mathbf{w}^\top \boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right)\right)^2$$

- Using a Gaussian prior, $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$, we have

$$\log \Pr(\mathbf{w}) = -\frac{1}{2\alpha} \|\mathbf{w}\|^2 + \text{const}$$

- Which leads to L_2 regularization:

$$\mathbf{w} = \arg\max_{\mathbf{w}} \log \Pr(\mathbf{w} \mid \mathcal{D}) = \left(\boldsymbol{\Psi}^\top \boldsymbol{\Psi} + \alpha \mathbf{I}\right)^{-1} \boldsymbol{\Psi}^\top \mathbf{t}$$

- In this case, we minimize

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} \left(t^{(i)} - \mathbf{w}^\top \boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right)\right)^2 + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}$$

- This is equivalent to solving

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{i=1}^{N}\left(\mathbf{w}^\top\boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right) - t^{(i)}\right)\boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right) = \sum_{i=1}^{N} a_i\boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right) = \boldsymbol{\Psi}^\top\mathbf{a}$$

  where the vector $\mathbf{a}$ has the ith entry

$$a_i = -\frac{1}{\lambda}\left(\mathbf{w}^\top\boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right) - t^{(i)}\right)$$

- Substitute $\mathbf{w} = \boldsymbol{\Psi}^\top\mathbf{a}$ back in $E(\mathbf{w})$ to get

$$E(\mathbf{a}) = \frac{1}{2}\mathbf{a}^\top\boldsymbol{\Psi}\boldsymbol{\Psi}^\top\boldsymbol{\Psi}\boldsymbol{\Psi}^\top\mathbf{a} - \mathbf{a}^\top\boldsymbol{\Psi}\boldsymbol{\Psi}^\top\mathbf{t} + \frac{1}{2}\mathbf{t}^\top\mathbf{t} + \frac{1}{2}\mathbf{a}^\top\boldsymbol{\Psi}\boldsymbol{\Psi}^\top\mathbf{a}$$

Kernel Ridge Regression

- To minimize $E(\mathbf{a})$, we introduce the gram matrix $\mathbf{K} = \boldsymbol{\Psi}\boldsymbol{\Psi}^\top$, i.e.

$$\mathbf{K}_{ij} = \boldsymbol{\Psi}\left(\mathbf{x}^{(i)}\right)^\top\boldsymbol{\Psi}\left(\mathbf{x}^{(j)}\right) \triangleq k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$$

  where we call k the **kernel**

- Therefore, we minimize

$$E(\mathbf{a}) = \frac{1}{2}\mathbf{a}^\top\mathbf{K}\mathbf{K}^\top\mathbf{a} - \mathbf{a}^\top\mathbf{K}\mathbf{t} + \frac{1}{2}\mathbf{t}^\top\mathbf{t} + \frac{1}{2}\mathbf{a}^\top\mathbf{K}\mathbf{a}$$

- Set the gradient of $E(\mathbf{a})$ to zero and solve for $\mathbf{a}$, we get

$$\mathbf{a} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{t}$$

- Substitute back into the linear regression model:

$$y(\mathbf{x}) = \mathbf{w}^\top\boldsymbol{\psi}(\mathbf{x}) = \mathbf{a}^\top\boldsymbol{\Psi}\boldsymbol{\psi}(\mathbf{x}) = \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{t}$$

  where $\mathbf{k}(\mathbf{x})$ is a vector with entries $k\left(\mathbf{x}^{(i)}, \mathbf{x}\right)$

- This is known as dual formulation, or the kernel trick
- The solution can be expressed as a linear conbination of the elements of $\boldsymbol{\psi}(\mathbf{x})$The prediction at $\mathbf{x}$ is give by a linear combination of the target values
- Dual formuation requires inverting an $N \times N$ matrix, whereas the standard one requires inverting an $M \times M$ matrix
  - M is the dimension of the feature space
  - N is the number of samples
- Advantage of the dual formulation: it is expressed entirely in terms of the kernel function

Kernels

- A symmetric matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ is **positive semidefinite (PSD)** if for every vector $\mathbf{u} \in \mathbb{R}^m$,

$$\mathbf{u}^\top\mathbf{A}\mathbf{u} \geq 0$$

- A **kernel** $k(\mathbf{x}, \mathbf{x}')$ is any function such that for any N data points $\mathbf{x}^{(i)}$ for $i = 1, \ldots, N$, the kernel matrix $\mathbf{K}$ with entries $\mathbf{K}_{ij} = k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$ is PSD
- In general, we use feature maps to define kernels:

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\psi}(\mathbf{x})^\top \boldsymbol{\psi}(\mathbf{x}')$$

- We could directly choose a kernel and work with it
- Kernels provide a measure of proximity between $\mathbf{x}$ and $\mathbf{x}'$

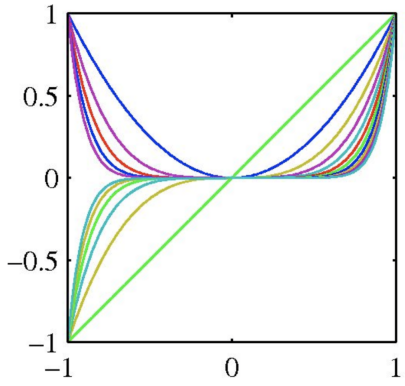
Feature Map as Kernel

- Let $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\psi}(\mathbf{x})^\top \boldsymbol{\psi}(\mathbf{x}')$ and $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ be inputs
- The kernel matrix is given as $K_{ij} = k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$
- We can show that this matrix is PSD

Local vs. Global Kernels

Polynomial basis functions:

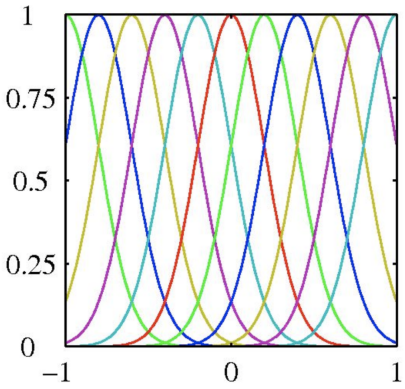
$$\phi_j(x) = x^j.$$



Basis functions are global: small changes in x affect all basis functions.

Gaussian basis functions:

$$\phi_j(x) = \exp\left(-\frac{(x - \mu_j)^2}{2s^2}\right).$$

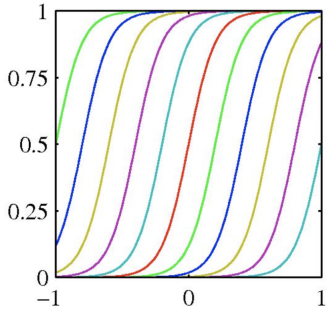


Basis functions are local: small changes in x only affect nearby basis functions.
μ_j and s control location and scale (width).

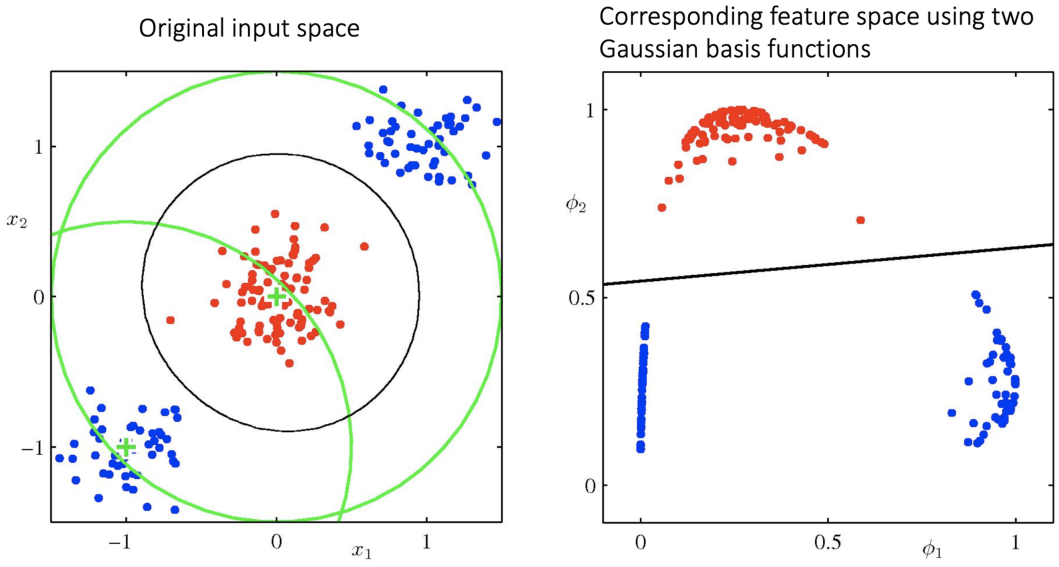
Radial Basis Functions

- Radial basis functions have the property that each basis function depends only on the radial distance from a center $\boldsymbol{\mu}_j$, i.e.

$$\boldsymbol{\psi}_j(\mathbf{x}) = h\left(\|\mathbf{x} - \boldsymbol{\mu}_j\|\right)$$



- Sigmoidal basis functions: h is sigmoid
- Gaussian basis functions: h is normal pdf
- E.g.



  - Define two Gaussian basis functions with centers shown by the green crosses, and with contours shown by the green circles
  - Linear decision boundary (right) is obtained by using logistic regression, and corresponds to the nonlinear decision boundary in the input space (left, black curve)
- Given a set of data samples $\left(\mathbf{x}^{(i)}, t^{(i)}\right)$ for $i = 1, \ldots, N$, we want to find a smooth function f such that

$$f(\mathbf{x}) \approx t$$

  - Achieved by expressing $f(\mathbf{x})$ as a linear combination of radial basis functions, one centered on every data point:

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i h\left(\left\|\mathbf{x} - \mathbf{x}^{(i)}\right\|\right)$$

    where w_i are found by least squares
- Similar to kNN

Neural Networks and Feature Learning

- Neural networks can be viewed as a way of learning features
- For the last layer:
  - If the task is regression, choose

$$\mathbf{y} = f^{(L)}\left(\mathbf{h}^{(L-1)}\right) = \left(\mathbf{w}^{(L)}\right)^{\top} \mathbf{h}^{(L-1)} + b^{(L)}$$

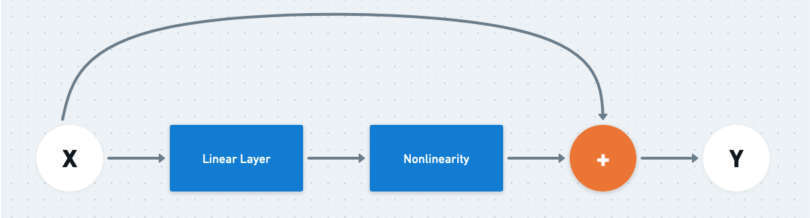
  - If the task is binary classification, choose

$$\mathbf{y} = f^{(L)}\left(\mathbf{h}^{(L-1)}\right) = \sigma\left[\left(\mathbf{w}^{(L)}\right)^{\top} \mathbf{h}^{(L-1)} + b^{(L)}\right]$$

13 Attention and Transformers

Residual Connections

- When we stack a large number of layers together, the signal may get squashed to 0, or blow up to ∞
- To reduce the effect of those problems, we propagate the signal to layers further downstream, using **residual connections**

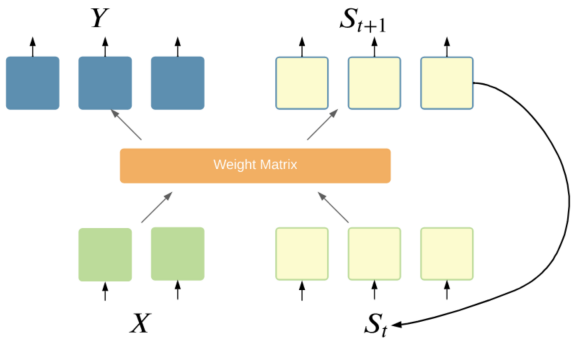


$$y = f(x; \theta) = \phi(Wx + b) + x$$

Recurrent Layers

- When modelling sequential data (e.g. text, time series), we can make the model stateful in order for it to help "carry" the information through the computation graph
- To do so, we add a state at timepoint t, s_t, and compute the output and the new state using some function

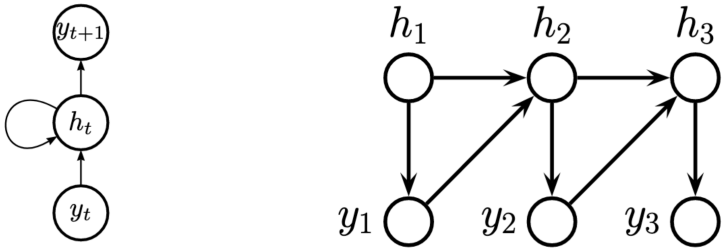
$$(y, s_{t+1}) = f(x, s_t)$$



- This is called a **recurrent layer**

Recurrent Neural Network (RNN)

- A **recurrent neural network** uses recurrent layers
- In the simplest possible option, the function $f(x, h)$ is a simple feed-forward neural network
- When training RNNs, each item in a sequence is used as input
- During inference, each item in the sequence will depend on previous predictions



Attention is All You Need

- We want to be able to look at a lot of the previous inputs at once
- Can score each of the hidden states by how well it is associated with the state we will be predicting
- The attention mechanism consists of 3 steps:
  1. Generate a score for each of the hidden states
  2. Apply the softmax function to the scores
  3. Multiply each of the hidden states by the output of the softmax and add them together
- To score each of the hidden states, we create 3 separate embeddings from each of our inputs, by multiplying them by learned matrices:

$$q = W^Q x$$
$$k = W^K x$$
$$v = W^V x$$

- Define the **attention layer** as

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

  where α is the scoring function

Dot Product Attention

- **Dot product attention**: obtain the scores by a normalized dot product of the k and q vectors:

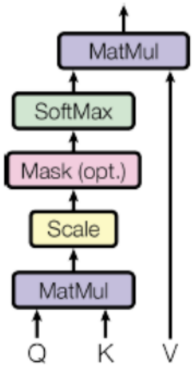
$$b(q, k) = \frac{q^\top k}{\sqrt{d}}$$

  where d is a normalizing constant, usually the dimensionality of the vectors
- Set our attention weights α_i to be the softmax of all the scores:

$$\alpha_i(q, k_i) = \frac{\exp(b(q, k_i))}{\sum_{j=1}^{m} \exp(b(q, k_j))}$$

- The entire process can be written as

$$Y = Attn(Q, K, V) = \sigma\left(\frac{QK^\top}{\sqrt{d}}\right) V = \sigma\left(\frac{W^Q X \left(W^K X\right)^\top}{\sqrt{d}}\right) W^V X$$



Connection to Kernels

- We compare the input x to each of the traning examples X using a kernel to get a vector of similarity scores

$$\alpha = \|K(x, x_i)\|_{i=1}^{m}$$

- Use this vector to retrieve a weighted combination of the corresponding target values y_i as

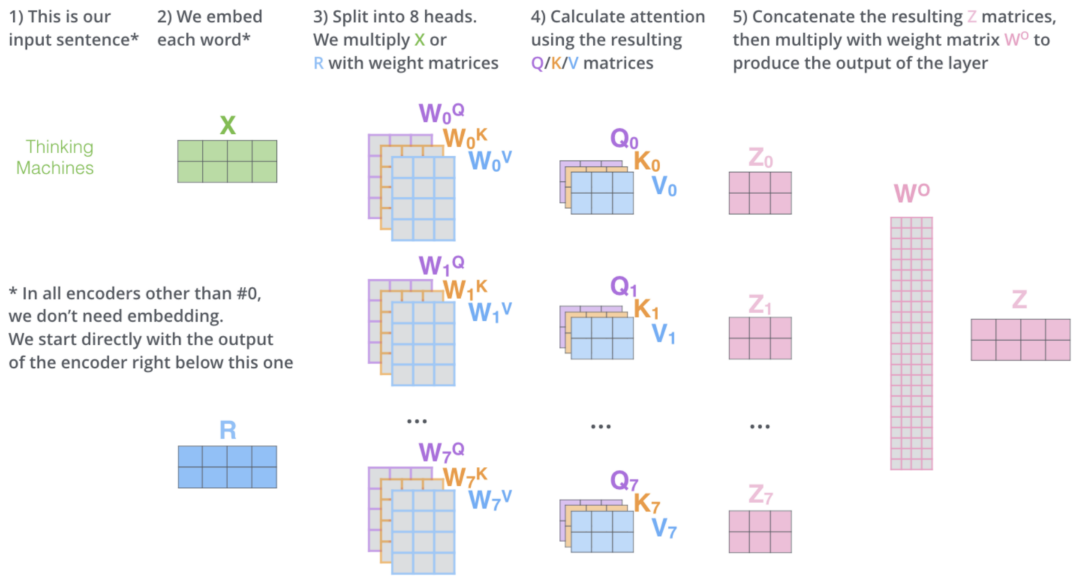
$$\hat{y} = \sum_{i=1}^{m} \alpha_i y_i$$

- If we replace the stored examples matrix X with a learned embedding $K = W^K X$, stored outputs with $V = W^V Y$, and create an input embedding $q = W^Q x$, then we get the same result as the dot product attention

Multi Head Attention

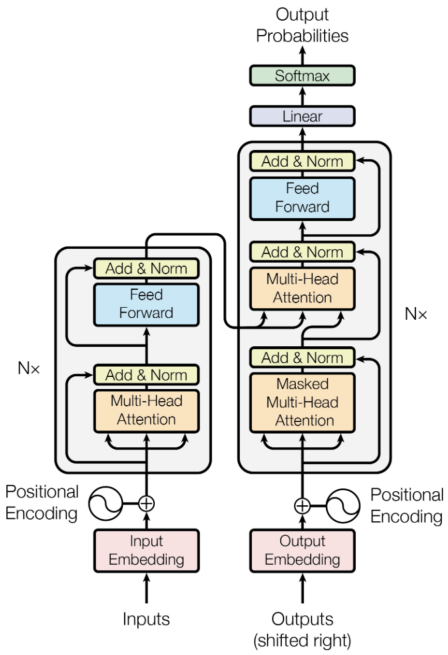
- We can have multiple *attention heads*, each with a different set of W^Q, W^K, W^V matrices
- We can concatenate the outputs of all the attention heads and multiplied by a learned matrix W^O
- This methos is called **multi head attention**

$$o = MHA(Q, K, V) = Concat(h_1, \ldots, h_h)W^O = Concat(Attn(Q_1, K_1, V_1), \ldots, Attn(Q_h, K_h, V_h))W^O$$



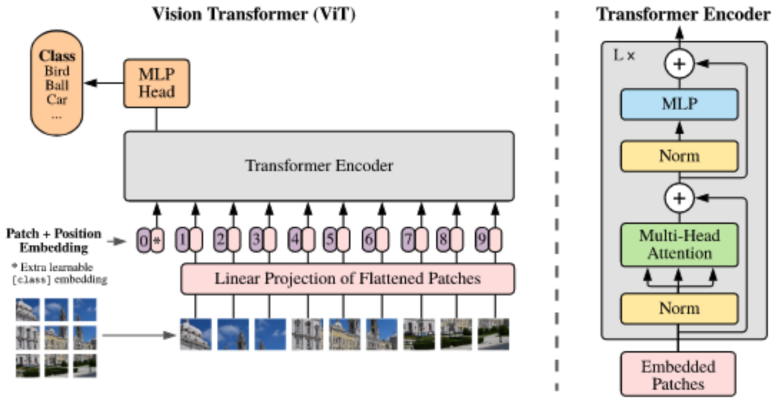
Transformer

- Consists of two stacks of blocks, the **encoder** and the **decoder**

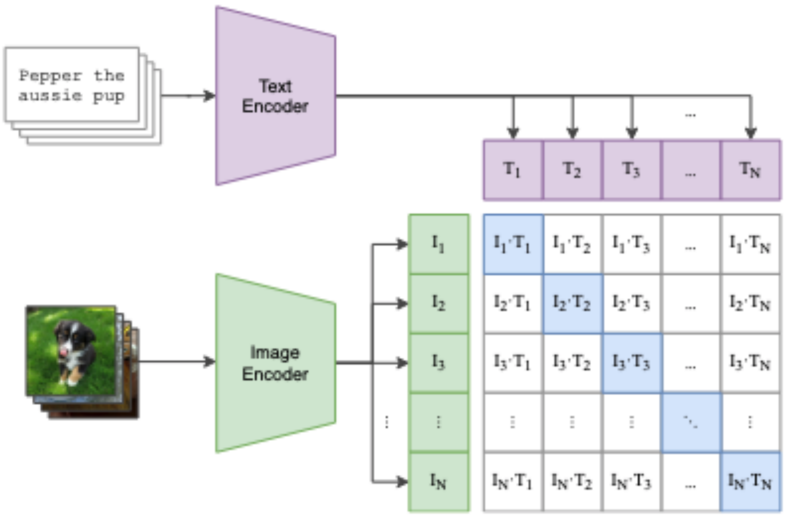


- The encoder is a stack of 6 blocks ($N = 6$), each of which is further split into two distinct sub-blocks
  - The first is a multi head self attention mechanism
  - The second is a simple feed forward NN
  - Both of the sub-blocks have a residual connection around them, followed by normalization
- The decoder is a stack of 6 blocks ($N = 6$). In addition to the 2 sub-blocks of the encoder, it has a third sub-block (in the middle)
  - This sub-block performs multi-head attention over the output of the encoder
  - This attention uses Q from the previous decoder layer, and K, V from the output of the encoder
- The input embedding is a learnable *token* embedding similar to Word2Vec
  - The *positional embedding* is either a learnable (representing position in a sequence) embedding, or a predefined embedding

Vision Transformers



CLIP



Contrastive Learning

- Given a dataset X, at each training step, pick samples x_i, v_i^+, v_i^-, and train an embedding function f_θ such that

$$f_\theta(x) \approx f_\theta\left(v_i^+\right)$$

and

$$f_\theta(x) \not\approx f_\theta\left(v_i^-\right)$$

- E.g.

$$\mathcal{L}\left(x, v^-, v^+\right) = \max\left(f(x)^\top f(v^-) - f(x)^\top f(v^+) + m, 0\right)$$

  - Max ensures that the term is nonnegative
  - The dot products refer to the cosine similarity

- E.g.

$$\mathcal{L}\left(x, v^-, v^+\right) = -\log\left[\frac{\exp\left(\frac{f(x)^\top f(v^+)}{\tau}\right)}{\exp\left(\frac{f(x)^\top f(v^+)}{\tau}\right) + \exp\left(\frac{f(x)^\top f(v^-)}{\tau}\right)}\right]$$

14 Gaussian Processes

Linear Regression

- Given a training set of inputs and targets $\left\{\left(\mathbf{x}^{(i)}, t^{(i)}\right)\right\}_{i=1}^{N}$, a linaer model is given by

$$\mathbf{y} = \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x})$$

  where $\boldsymbol{\psi}(\mathbf{x})$ is the feature map

- When vectorized, we have the design matrix $\mathbf{X}$ in the input space and

$$\boldsymbol{\Psi} = \begin{bmatrix} - & \boldsymbol{\psi}(\mathbf{x}^{(1)}) & - \\ - & \boldsymbol{\psi}(\mathbf{x}^{(2)}) & - \\ & \cdots & \\ - & \boldsymbol{\psi}(\mathbf{x}^{(N)}) & - \end{bmatrix}$$

  to generate predictions

$$\mathbf{y} = \boldsymbol{\Psi}\mathbf{w}$$

- Can be probabilistically interpreted by assuming a Gaussian noise model

$$t \mid \mathbf{x} \sim \mathcal{N}\left(\mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x}), \sigma^2\right)$$

  and a Gaussian prior

$$\mathbf{w} \sim \mathcal{N}\left(\mathbf{0}, \alpha^{-1}\mathbf{I}\right)$$

Distribution over Prediction Function

- In practice, we evaluate the prediction function $y(\mathbf{x})$ at specific points, e.g. at the training data points $\mathbf{x}^{(i)}$ for $i = 1, \ldots, N$
- We are interested in the joint distribution of the function values $y\left(\mathbf{x}^{(1)}\right), \ldots, y\left(\mathbf{x}^{N}\right)$, which we denote by the vector $\mathbf{y}$
- We have

$$\mathbf{y} = \boldsymbol{\Psi}\mathbf{w} \qquad \mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$$

- Thus

$$\mathbf{y} \sim \mathcal{N}(\mathbf{0}; \mathbf{K}) \qquad \mathbf{K} = \frac{1}{\alpha}\boldsymbol{\Psi}\boldsymbol{\Psi}^\top$$

  where $\mathbf{K}$ is the (scaled) Gram matrix

$$\mathbf{K}_{ij} = \frac{1}{\alpha}k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \frac{1}{\alpha}\boldsymbol{\psi}\left(\mathbf{x}^{(i)}\right)^\top \boldsymbol{\psi}\left(\mathbf{x}^{(j)}\right)$$

Gaussian Process

- A **Gaussian process** is a probability distribution over functions $y(\mathbf{x})$ such that any set of values $y(\mathbf{x})$ evaluated at an arbitrary set of points is jointly Gaussian
- The case $\mathbf{x} \in \mathbb{R}^2$ is called Gaussian random field
- The joint distribution is specified completely by the second-order statistics, i.e. the mean and the covariance
- In most applications, the mean of $y(\mathbf{x})$ is set to zero

- Which makes the Gaussian process completely specified by the covariance

$$\mathbb{E}\left[y\left(\mathbf{x}^{(i)}\right) y\left(\mathbf{x}^{(j)}\right)\right] = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)$$

- We can directly define the kernel of a Gaussian process (without the need for a feature map)

Gaussian Process for Regression

- We have the linear model

$$t \mid \mathbf{x} \sim \mathcal{N}\left(y(\mathbf{x}), \sigma^2\right) \qquad y(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x})$$

- Given N samples, we have

$$\mathbf{t} \mid \mathbf{y} \sim \mathcal{N}\left(\mathbf{y}, \sigma^2 \mathbf{I}\right)$$

- Since $\mathbf{y}$ is a Gaussian process, we have

$$\mathbf{y} \sim \mathcal{N}(\mathbf{0}; \mathbf{K})$$

- Therefore the marginal of $\mathbf{t}$ is given by

$$\mathbf{t} \sim \mathcal{N}(\mathbf{0}, \mathbf{C}) \qquad \mathbf{C} = \mathbf{K} + \sigma^2 \mathbf{I}$$

  where

$$C\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) + \sigma^2 \delta_{ij}$$

  - $\delta_{ij} = 1$ if $i = j$ and 0 otherwise

- Define $\mathbf{t}_N = \left(t^{(1)}, t^{(2)}, \ldots, t^{(N)}\right)^\top$
- We have the marginal of $\mathbf{t}_N$ given by

$$\mathbf{t}_N \sim \mathcal{N}(0, \mathbf{C}_N) \qquad \mathbf{C}_N = \mathbf{K}_N + \sigma^2 \mathbf{I}$$

  where

$$C_N\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) + \sigma^2 \delta_{ij}$$

- This reflects the two Gaussian sources of randomness (i.e. $\mathbf{K}_N$ and $\sigma^2 \mathbf{I}$)
- Goal: predict for a new input $\mathbf{x}^{(N+1)}$
- Need $\Pr\left(t^{(N+1)} \mid \mathbf{t}_N\right)$
  - Note that $\mathbf{x}^{(i)}$s are treated as constants
- We have

$$\mathbf{t}_{N+1} \sim \mathcal{N}(0, \mathbf{C}_{N+1}) \qquad \mathbf{C}_{N+1} = \mathbf{K}_{N+1} + \sigma^2 \mathbf{I}$$

  where

$$C_{N+1}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) + \sigma^2 \delta_{ij}$$

$$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C}_N & \mathbf{k} \\ \mathbf{k}^\top & c \end{bmatrix}$$

  - c is a scalar with

$$c = \frac{1}{\alpha} k\left(\mathbf{x}^{(N+1)}, \mathbf{x}^{(N+1)}\right) + \sigma^2$$

  - $\mathbf{k}$ is a vector with entries

$$k_i = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(N+1)}\right)$$

- Since $\mathbf{t}_{N+1}$ is multivariate Gaussian, we can easily find $t^{(N+1)} \mid \mathbf{t}_N$
- Property of multivariate Gaussian distribution
  - If we have $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \qquad \boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}$$

  - Then

$$\mathbf{x}_1 \mid (\mathbf{x}_2 = \mathbf{a}) \sim \mathcal{N}(\mathbf{m}, \mathbf{C})$$

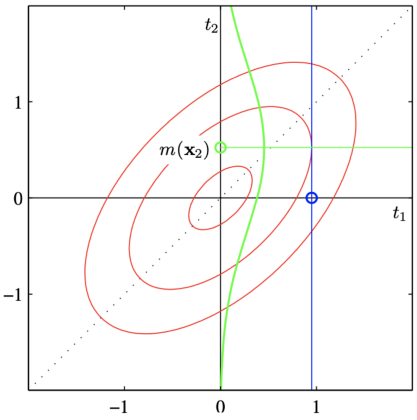
    with

$$\mathbf{m} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{a} - \boldsymbol{\mu}_2) \qquad \mathbf{C} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}$$

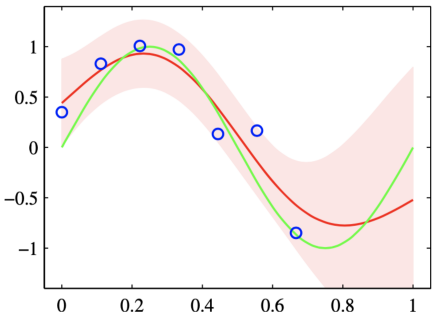
- Since $\mathbf{t}_{N+1}$ is multivariate Gaussian, $t^{(N+1)} \mid \mathbf{t}_N$ is also Gaussian with mean and variance

$$m\left(\mathbf{x}^{(N+1)}\right) = \mathbf{k}^\top \mathbf{C}_N^{-1} \mathbf{t}_N \qquad \sigma^2\left(\mathbf{x}^{(N+1)}\right) = c - \mathbf{k}^\top \mathbf{C}_N^{-1} \mathbf{k}$$

- The vector $\mathbf{k}$ is a function of the new test input $\mathbf{x}^{(N+1)}$
- The predictive distribution is a Gaussian whose mean and variance both depend on $\mathbf{x}^{(N+1)}$



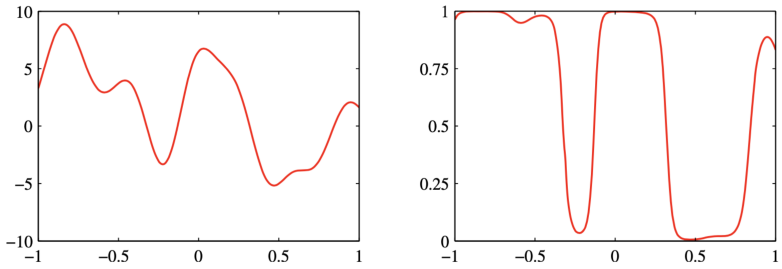
- One training point (blue) and one test point (green)
- Red ellipses show contours of the joint $\Pr\left(t^{(1)}, t^{(2)}\right)$
- Conditioning on $t^{(1)}$ corresponds to the vertical blue line
- $\Pr\left(t^{(2)} \mid t^{(1)}\right)$ is shown by the green curve



- The green curve is the true sinusoidal function
- The datapoints are sampled from the true function plus some Gaussian noise
- The red line shows the mean of the Gaussian process predictive distribution
- The shaded region corresponds to two standard deviations

Gaussian Process for Classification

- Consider a classification problem with target variables $t \in \{0, 1\}$
- Define a Gaussian process over a function $a(\mathbf{x})$ and then transform the function using sigmoid $y(\mathbf{x}) = \sigma(a(\mathbf{x}))$
- We obtain a non-Gaussian stochastic process over functions $y(\mathbf{x}) \in \{0, 1\}$



  - Left: $a(\mathbf{x})$, right: $y(\mathbf{x})$

- The probability distribution over target is given by

$$\Pr(t \mid a) = \sigma(a)^t (1 - \sigma(a))^{1-t}$$

- Need to compute

$$\Pr\left(t^{(N+1)} \mid \mathbf{t}_N\right)$$

  - Notice that this won't be Gaussian since $y(\mathbf{x})$ is not a Gaussian process (only $a(\mathbf{x})$ is a Gaussian process)

- We have

$$\mathbf{a}_{N+1} \sim \mathcal{N}(\mathbf{0}, \mathbf{C}_{N+1})$$

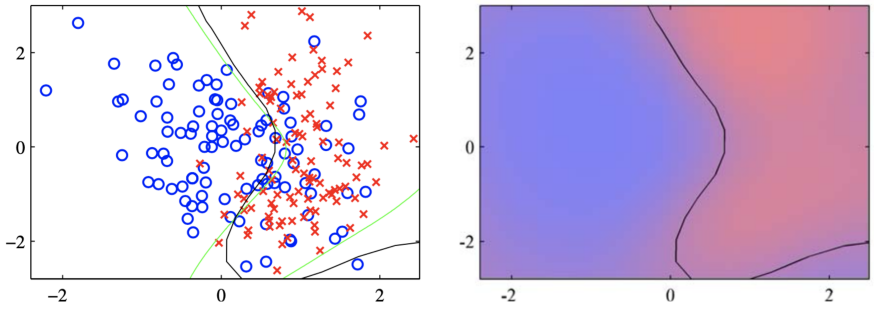
  where

$$C_{N+1}\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \frac{1}{\alpha} k\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) + \nu \delta_{ij}$$

- To find $\Pr\left(t^{(N+1)} \mid \mathbf{t}_N\right)$, we compute

$$\Pr\left(t^{(N+1)} \mid \mathbf{t}_N\right) = \int \Pr\left(t^{N+1} \mid \mathbf{a}_{N+1}\right) \Pr(\mathbf{a}_{N+1} \mid \mathbf{t}_N) d\mathbf{a}_{N+1}$$

  - Intractable to compute
  - Can rely on MCMC-based methods, or numerical integration to approximate this integral



- Leftt: optimal decision boundary from the true distribution in green, decision boundary from the Gaussian process classifier in black
- Right: predicted posterior for the blue and red classes together with the Gaussian process decision boundary

Learning the Hyperparameters

- We didn't do any learning other than choosing a kernel
- Rather than fixing the covariance function, we may prefer to use a parametric family of functions and then infer the parameter values from the data
- Denoting the hyperparameters with θ, the likelihood of the Gaussian process model is

$$\log \Pr(\mathbf{t} \mid \theta) = -\frac{1}{2} \log \|\mathbf{C}_N\| - \frac{1}{2} \mathbf{t}^\top \mathbf{C}_N^{-1} \mathbf{t} - \frac{N}{2} \log(2\pi)$$

- Can use gradient-based optimization