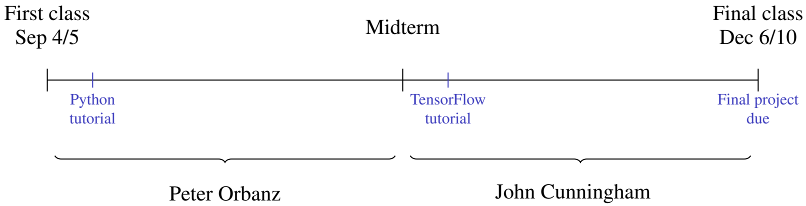
COURSE ADMIN

Term Timeline



First class
Sep 4/5

Midterm

Final class
Dec 6/10

Python
tutorial

TensorFlow
tutorial

Final project
due

Peter Orbanz

John Cunningham

Dates

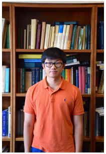
Python tutorial	6/10 September
Midterm exam	18/22 October
TensorFlow tutorial	24/25 October
Final project due	10 December

Assistants and Grading

Teaching Assistants

Andrew Davison

Ian Kinsella



Peter Lee

Gabriel Loaiza

Office Hours	Mon/Tue 5:30-7:30pm, Room 1025, Dept of Statistics, 10th floor SSW

Class Homepage

`https://www.adavison.co.uk/teaching/AdvancedML18/`

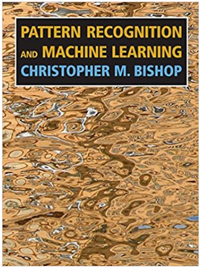
Homework

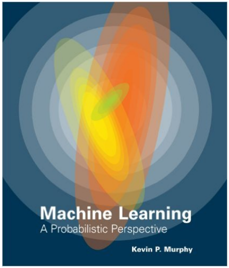
- Some homework problems and final project require coding
- Coding: Python
- Homework due: Tue/Wed at 4pm – no late submissions, please
- You can drop two homeworks from your final score

READING

The relevant course material are the slides.

Books (optional)





See class homepage for references.

TOPICS (TENTATIVE)

Part I (Orbanz)

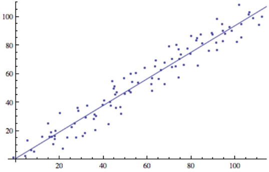
- Markov and hidden Markov models
- Graphical models
- Sampling and MCMC algorithms
- Variational inference
- Neural network basics

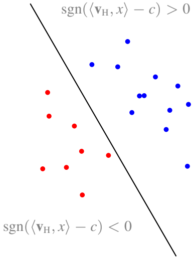
Part II (Cunningham)

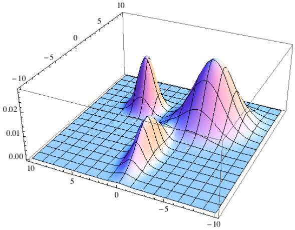
- NN software
- Neural networks: convolutional, recurrent, etc.
- Reinforcement learning
- Dimension reduction and autoencoders

INTRODUCTION

Recall Previous Term




$\mathrm{sgn}(\langle \mathbf{v}_{\mathrm{H}}, x\rangle - c) > 0$
$\mathrm{sgn}(\langle \mathbf{v}_{\mathrm{H}}, x\rangle - c) < 0$



	Supervised learning	Unsupervised learning
Problems	Classification Regression	Clustering (mixture models) HMMs Dimension reduction (PCA)
Solutions	Functions	Distributions

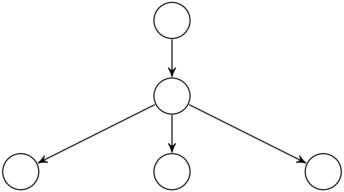
OUR MAIN TOPICS

Neural networks

- Define *functions*
- Represented by directed graph

Graphical models

- Define *distributions*
- Represented by directed graph



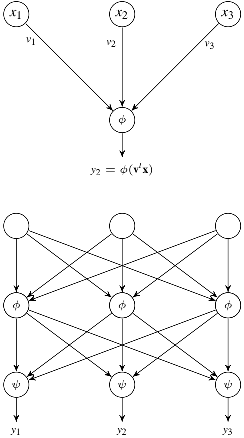
Neural networks

- Each vertex represents a function
- Incoming edges: Function arguments
- Outgoing edges: Function values
- Learning: Differentiation/optimization

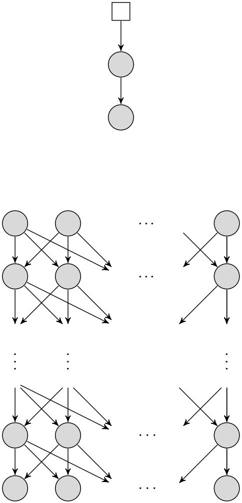
Graphical models

- Each vertex represents a distribution
- Incoming edges: Conditions
- Outgoing edges: Draws from distribution
- Learning: Estimation/inference

Neural networks



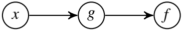
Graphical models



NNs and Graphical Models

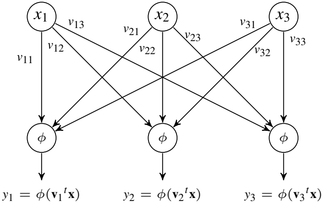
Neural networks

- Representation of function using a graph
- Layers:



- Symbolizes: $f(g(x))$

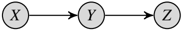
"f depends on x only through g"



x_1
x_2
x_3
v_{11}
v_{12}
v_{13}
v_{21}
v_{22}
v_{23}
v_{31}
v_{32}
v_{33}
ϕ
ϕ
ϕ
$y_1 = \phi(\mathbf{v}_1{}^t\mathbf{x})$
$y_2 = \phi(\mathbf{v}_2{}^t\mathbf{x})$
$y_3 = \phi(\mathbf{v}_3{}^t\mathbf{x})$

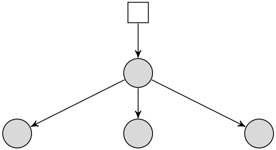
Graphical models

- Representation of a distribution using a graph
- Layers:



- Symbolizes: $p(x, z, y) = p(z|y)p(y|x)p(x)$

"Z is conditionally independent of X given Y"



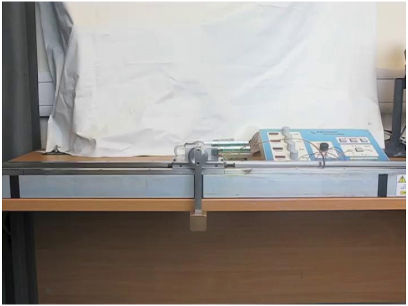
Topics (tentative)

Part I (Orbanz)

- Markov and hidden Markov models
- Graphical models
- Sampling and MCMC algorithms
- Variational inference
- Neural network basics

Part II (Cunningham)

- NN software
- Neural networks: convolutional, recurrent, etc.
- Reinforcement learning
- Dimension reduction and autoencoders



Task

Balance the pendulumn upright by moving the sled left and right.

- The computer can control *only* the motion of the sled.
- Available data: Current state of system (measured 25 times/second).

LEARNING AND STATISTICS

Formalization

$$\begin{array}{ccl} \text{State} & = & \text{4 variables (sled location, sled velocity, angle, angular velocity)} \\ \text{Actions} & = & \text{sled movements} \end{array}$$

The system can be described by a function

$$\begin{aligned} f : \quad \mathcal{S} \times \mathcal{A} &\to \mathcal{S} \\ (\text{state}, \text{action}) &\mapsto \text{state} \end{aligned}$$

LEARNING AND STATISTICS

LEARNING AND STATISTICS

After each run

Fit a function

$$\begin{aligned} f: \quad \mathcal{S} \times \mathcal{A} &\to \mathcal{S} \\ (\text{state}, \text{action}) &\mapsto \text{state} \end{aligned}$$

to the data obtained in previous runs.

Running the system involves:

1. The function f, which tells the system “how the world works”.
2. An optimization method that uses f to determine how to move towards the optimal state.

Note well

Learning how the world works is a regression problem.

SEQUENTIAL DATA AND MARKOV MODELS

MOTIVATION: PAGERANK

Simple random walk

Start with a graph G. Define a random sequence of vertices as follows:

- Choose a vertex X_1 uniformly at random.
- Choose a vertex X_2 uniformly at random from the neighbors of X_1. Move to X_2.
- Iterate: At step n, uniformly sample a neighbor X_n of X_{n-1}, and move to X_n.

This is called *simple random walk* on G.

Google's PageRank Algorithm

To sort the web pages matching a search query by importance, PageRank:

1. Defines a graph G whose vertices are web pages and whose edges are web links.
2. Computes the probability distribution on vertices x in G given by

   $$P_n(x) = P(X_n = x) \qquad \text{where} \qquad X_1, \dots X_n \text{ is a simple random walk on } G$$

   and n is very large.

We will try to understand (a) why and (b) how P_n can be computed.

SEQUENTIAL DATA

So far: I.i.d. sequences

We have assumed that samples are of the form

$$X_1 = x_1, X_2 = x_2, \ldots \qquad \text{where} \qquad X_1, X_2, \ldots \sim_{\text{iid}} P$$

for some distribution P. In particular, the order of observations does not matter.

Now: Dependence on the past

We now consider sequences in which the random variable X_n can be stochastically dependent on $X_1, \ldots, X_{n-1}$, so we have to consider conditional probabilities of the form

$$P(X_n = x_n | X_1 = x_1, \ldots, X_{n-1} = x_{n-1}) \ .$$

Application examples

- Speech and handwriting recognition.
- Time series, e.g. in finance. (These often assume a *continuous* index. Our index n is discrete.)
- Simulation and estimation algorithms (Markov chain Monte Carlo).
- Random walk models (e.g. web search).

MARKOV MODELS

Markov models

The sequence $(X_n)_n$ is called a **Markov chain of order** r if X_n depends only on a fixed number r of previous samples, i.e. if

$$P(X_n = x_n | X_1 = x_1, \ldots, X_{n-1} = x_{n-1}) \; = \; P(X_n = x_n | X_{n-r} = x_{n-r}, \ldots, X_{n-1} = x_{n-1})$$

If we simply call $(X_n)_n$ a **Markov chain**, we imply $r = 1$.

Initial state

The first state in the sequence is special because it does not have a "past", and is usually denoted X_0.

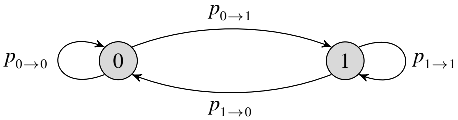
Example: $r = 2$

$$\underbrace{X_0 = x_0,\ X_1 = x_1,}_{\substack{X_4 \text{ is independent} \\ \text{of these } \textit{given } X_2,\, X_3}}\ \underbrace{X_2 = x_2,\ X_3 = x_3,}_{X_4 \text{ may depend on these}}\ X_4 = ?$$

GRAPHICAL REPRESENTATION

A simple binary chain

Suppose $\mathbf{X} = \{0, 1\}$.



- We regard 0 and 1 as possible "states" of X, represented as vertices in a graph.
- Each pair $X_{n-1} = s, X_n = t$ in the sequence is regarded as a "transition" from s to t and represented as an edge in the graph.
- Each edge $s \to t$ is weighted by the probability

$$p_{s\to t} := P(X_n = t | X_{n-1} = s) \ .$$

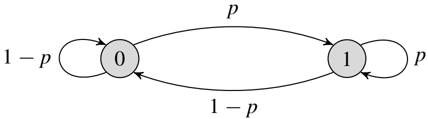
State space

The elements of the sample space $\mathbf{X}$ are called the **states** of the chain. $\mathbf{X}$ is often called the **state space**. We generally assume that $\mathbf{X}$ is finite, but Markov chains can be generalized to infinite and even uncountable state spaces.

GRAPHICAL REPRESENTATION

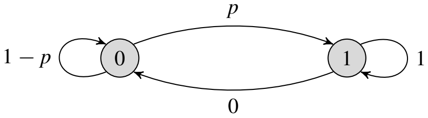
First example: Independent coin flips

Suppose X is a biased coin with $P(X_n = 1) = p$ independently of X_{n-1}. In other words, the sequence (X_n) is iid Bernoulli with parameter p.



Breaking independence

Here is a simple modification to the chain above; only $p_{1\to 0}$ and $p_{1\to 1}$ have changed:



This is still a valid Markov chain, but the elements of the sequence are no longer independent.

GRAPHICAL REPRESENTATION

Observation

The graph representation is only possible if $p_{s \rightarrow t}$ is independent of n. Otherwise we would have to draw a different graph for each n.

If $p_{s \rightarrow t}$ does not depend on n, the Markov chain is called **stationary**.

Transition matrix

The probabilities $p_{s \rightarrow t}$ are called the **transition probabilities** of the Markov chain. If $|\mathbf{X}| = d$, the $d \times d$-matrix

$$\mathbf{p} := (p_{i \rightarrow j})_{j,i \leq d} = \begin{pmatrix} p_{1\rightarrow 1} & \cdots & p_{d \rightarrow 1} \\ \vdots & & \vdots \\ p_{1 \rightarrow d} & \cdots & p_{d \rightarrow d} \end{pmatrix}$$

is called the **transition matrix** of the chain. This is precisely the adjacency matrix of the graph representing the chain. Each column is a probability distribution on d events.

Graphical Representation

Complete description of a Markov chain

The transition matrix does not completely determine the chain: It determines the probability of a state given a previous state, but not the probability of the starting state. We have to additionally specify the distribution of the first state.

Initial distribution

The distribution of the first state, i.e. the vector

$$P_{\text{init}} := (P(X_0 = 1), \dots, P(X_0 = d)) \ ,$$

is called the **initial distribution** of the Markov chain.

Representing stationary Markov chains

Any stationary Markov chain with finite state space can be completely described by a transition matrix $\mathbf{p}$ and an initial distribution P_{init}. That is, the pair $(\mathbf{p}, P_{\text{init}})$ completely determines the joint distribution of the sequence $(X_0, X_1, \dots)$.

Simple random walk

Suppose we are given a directed graph G (with unweighted edges). We had already mentioned that the **simple random walk** on G is the vertex-valued random sequence $X_0, X_1, \ldots$ defined as:

- We select a vertex X_0 in G uniformly at random.
- For $n = 1, 2, \ldots$, select X_n uniformly at random from the children of X_{n-1} in the graph.

Markov chain representation

Clearly, the simple random walk on a graph with d vertices is a Markov chain with

$$P_{\text{init}} = \left(\frac{1}{d}, \ldots, \frac{1}{d}\right) \qquad \text{and} \qquad p_{i\to j} = \begin{cases} \frac{1}{\text{\# edges out of } i} & \text{if } i \text{ links to } j \\ 0 & \text{otherwise} \end{cases}$$

RANDOM WALKS AND MARKOV CHAINS

Generalizing simple random walk

We can generalize the idea of simple random walk by substituting the uniform distributions by other distributions. To this end, we can weight each edge in the graph by a probability of following that edge.

Adjacency matrix

If the edge weights are proper probabilities, each row of the adjacency matrix must sum to one. In other words, the matrix is the transition matrix of a Markov chain.

Random walks and Markov chains

If we also choose a general distribution for the initial state of the random walk, we obtain a completely determined Markov chain. Hence:

Any Markov chain on a finite state space is a random walk on a weighted graph and vice versa.

INTERNET SEARCH

Queries

The first step in internet search is query matching:

1. The user enters a search query (a string of words).
2. The search engine determines all web pages indexed in its database which match the query.

This is typically a large set. For example, Google reports ca 83 million matches for the query "random walk".

The ranking problem

- For the search result to be useful, the most useful link should with high probability be among the first few matches shown to the user.
- That requires the matching results to be *ranked*, i.e. sorted in order of decreasing "usefulness".

POPULARITY SCORING

Available data

Using a web crawler, we can (approximately) determine the link structure of the internet. That is, we can determine:

- Which pages there are.
- Which page links which.

A web crawler cannot determine:

- How often a link is followed.
- How often a page is visited.

Web graph

The link structure can be represented as a graph with

vertices = web pages   and   edges = links.

RANDOM WALK NETWORK MODELS

Key idea

The popularity of a page x is proportional to the probability that a "random web surfer" ends up on page x after a n steps.

Probabilistic model

The path of the surfer is modeled by a random walk on the web graph.

Modeling assumptions

Two assumptions are implicit in this model:

1. Better pages are linked more often.
2. A link from a high-quality page is worth more than one from a low-quality page.

Remarks

- We will find later that the choice of n does not matter.
- To compute the popularity score, we first have to understand Markov chains a bit better.

STATE PROBABILITIES

Probability after $n = 1$ steps

If we *know* the initial state, then

$$P(X_1 = s_1 \mid X_0 = s_0) = p_{s_0 \to s_1} \ .$$

P_1 describes the probability of X_1 if we do *not* know the starting state (i.e. the probability before we start the chain):

$$\begin{aligned} P_1(s_1) = P(X_1 = s_1) &= \sum_{s_0 \in \mathbf{X}} P(X_1 = s_1 \mid X_0 = s_0) P_{\text{init}}(s_0) \\ &= \sum_{s_0 \in \mathbf{X}} p_{s_0 \to s_1} P_{\text{init}}(s_0) \ . \end{aligned}$$

Matrix representation

Recall that $\mathbf{p}$ is a $d \times d$-matrix and P_{init} a vector of length d. The equation for P_1 above is a matrix-vector product, so

$$P_1 = \mathbf{p} \cdot P_{\text{init}} \ .$$

STATE PROBABILITIES

Probability after $n = 2$ steps

The same argument shows that P_2 is given by

$$P_2(s_2) = \sum_{s_1 \in \mathbf{X}} p_{s_1 \rightarrow s_2} P_1(s_1) \ ,$$

hence

$$P_2 = \mathbf{p} \cdot P_1 = \mathbf{p} \cdot \mathbf{p} \cdot P_{\text{init}} \ .$$

For arbitary n

$$P_n = \mathbf{p}^n P_{\text{init}}$$

LIMITS AND EQUILIBRIA

Limiting distribution

Instead of considering P_n for a specific, large n, we take the limit

$$P_\infty := \lim_{n\to\infty} P_n = \lim_{n\to\infty} \mathbf{p}^n P_{\text{init}} ,$$

provided that the limit exists.

Observation

If the limit P_∞ exists, then

$$\mathbf{p} \cdot P_\infty = \mathbf{p} \cdot \lim_{n\to\infty} \mathbf{p}^n P_{\text{init}} = \lim_{n\to\infty} \mathbf{p}^n P_{\text{init}} = P_\infty ,$$

which motivates the next definition.

Equilibrium distribution

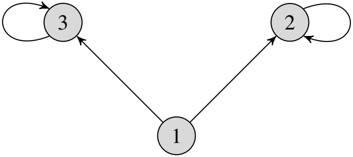
If $\mathbf{p}$ is the tansition matrix of a Markov chain, a distribution P on $\mathbf{X}$ which is invariant under $\mathbf{p}$ in the sense that

$$\mathbf{p} \cdot P = P$$

is called an **equilibrium distribution** or **invariant distribution** of the Markov chain.

WHAT CAN GO WRONG?

Problem 1: The equilibrium distribution may not be unique



For this chain, both $P = (0, 1, 0)$ and $P' = (0, 0, 1)$ are valid equilibria. Which one emerges depends on the initial state and (if we start in state 1) on the first transition.

Remedy

Require that there is a path in the graph (with non-zero probability) from each state to every other state. A Markov chain satisfying this condition is called **irreducible**.

WHAT CAN GO WRONG?

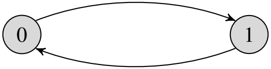
Recall that a sequence in $\mathbb{R}$ does not have a limit if it "oscillates". For example,

$$\lim_n 1^n = 1 \qquad \text{but} \qquad \lim_n (-1)^n \text{ does not exist}$$

Problem 2: The limit may not exist

- The chain on the right has no limit distribution.
- If we start e.g. in state 0, then:
  - 0 can only be reached in even steps.
  - 1 only in odd steps.
- The distribution P_n oscillates between

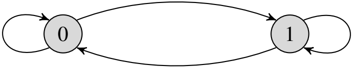
$$P_{\text{even}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \text{and} \qquad P_{\text{odd}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \ .$$



What Can Go Wrong?

Remedy

To prevent this (particular) problem, we can add two edges:



Now each state is reachable in every step.

The problem (at least this example) is that we have to leave the state before we can return to it. We prevent this, we introduce the following definition.

Aperiodic chains

We call a stationary Markov chain **aperiodic** if, for every state *s*,

$$P(X_n = s \,|\, X_{n-1} = s) = p_{s \to s} > 0 \ .$$

In short, a stationary chain is aperiodic if the transition matrix has non-zero diagonal.

EQUILIBRIUM DISTRIBUTIONS

We have introduced two definitions which prevent two rather obvious problems. Surprisingly, these definitions are all we need to guarantee limits.

Theorem

Suppose a Markov chain $(\mathbf{p}, P_{\text{init}})$ is stationary, and for each state $s \in \mathbf{X}$:

1. There is a path (with non-zero probability) from s to every other state (i.e. the chain is irreducible).
2. $p_{s \to s} > 0$ (i.e. the chain is aperiodic).

Then:

- The limit distribution P_∞ exists.
- The limit distribution is also the equlibrium distribution.
- The equilibrium distribution is unique.

COMPUTING THE EQUILIBRIUM

Power method

If the the transition matrix **p** makes the chain irreducible and aperiodic, we know that

equilibrium distribution = limit distribution .

This means we can approximate the equilibrium P_∞ by P_n: We start with any distribution P_{init} (e.g. uniform) and repeatedly multiply by **p**:

$$P_{n+1} = \mathbf{p} \cdot P_n$$

We can threshold the change between steps, e.g. by checking $\|P_{n+1} - P_n\| < \tau$ for some small τ.

Remark: Eigenstructure

The power method can be regarded as an eigenvector computation. The definition

$$P = \mathbf{p} \cdot P$$

of the equilibrium means that $P = P_\infty$ is an eigenvector of **p** with eigenvalue 1. If **p** is irreducible and aperiodic, it can be shown that 1 is the largest eigenvalue.

Constructing the transition matrix

We start with the web graph and construct the transition matrix of simple random walk, i.e.

$$a_{ij} := \begin{cases} \frac{1}{\text{\# edges out of } i} & \text{if } i \text{ links to } j \\ 0 & \text{otherwise} \end{cases}$$

A chain defined by $A := (a_{ij})$ will almost certainly not be irreducible (think of web pages which do not link anywhere). We therefore regularize A by defining

$$\mathbf{p} := (1-\alpha)A + \frac{\alpha}{d}\begin{pmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ 1 & \cdots & 1 \end{pmatrix}$$

for some small $\alpha \in (0,1)$.
Clearly, this makes $\mathbf{p}$ both irreducible and aperiodic.

Computing the equilibrium

Given $\mathbf{p}$, the equilibrium distribution is computed using the power method. Since the web changes, the power method can be re-run every few days with the previous equilibrium as initial distribution.

The Random Surfer Again

We can now take a more informed look at the idea of a random web surfer:

- Suppose the surfer is more likely to start on a popular page than on an unpopular one.
- In terms of the popularity model, this means

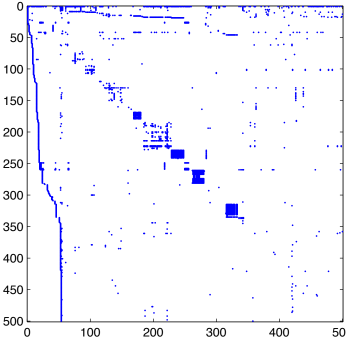
  $$X_0 \sim P_{\text{equ}} \;,$$

  where P_{equ} is the equilibrium distribution of the chain.
- After following any number of links n (with probabilities given by the transition matrix $\mathbf{p}$),

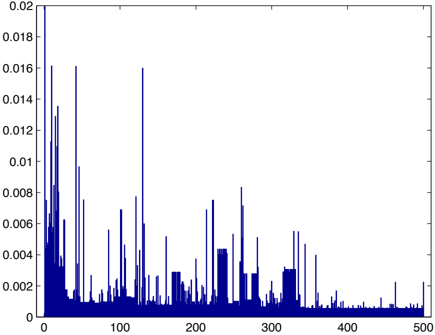
  $$P_n = \mathbf{p}^n P_{\text{equ}} = P_{\text{equ}} \;.$$

- In this sense, P_{equ} is really the consistent solution to our problem, even if we *compute* it by starting the random walk from e.g. a uniform initial distribution instead.
- In particular, it does not matter how we choose n in the model.

EXAMPLE



Adjacence matrix of the web graph of 500 web pages. The root (index 0) is www.harvard.edu.



Equilibrium distribution computed by PageRank.

See K. Murphy, "Machine Learning", MIT Press 2012.

Graphical Models

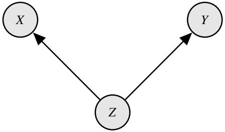
GRAPHICAL MODELS

A graphical model represents the dependence structure within a set of random variables as a graph.

Overview

Roughly speaking:

- Each random variable is represented by vertex.
- If Y depends on X, we draw an edge $X \to Y$.
- For example:



This says: "X depends on Z, and Y depends on Z".

- We have to be careful: The above does not imply that X and Y are independent. We have to make more precise what *depends on* means.

We will use the notation:

$$\begin{aligned} \mathcal{L}(X) =& \text{ distribution of the random variable } X \\ \mathcal{L}(X|Y) =& \text{ conditional distribution of } X \text{ given } Y \end{aligned}$$

($\mathcal{L}$ means “law”.)

Reason

- If X is discrete, $\mathcal{L}(X)$ is usually given by a mass function $P(x)$.
- If it is continuous, $\mathcal{L}(X)$ is usually given by a density $p(x)$.
- With the notation above, we do not have to distinguish between discrete and continuous variables.

DEPENDENCE AND INDEPENDENCE

Dependence between random variables $X_1, \ldots, X_n$ is a property of their joint distribution $\mathcal{L}(X_1, \ldots, X_n)$.

Recall

Two random variables are *stochastically independent*, or *independent* for short, if their joint distribution factorizes:

$$\mathcal{L}(X, Y) = \mathcal{L}(X)\mathcal{L}(Y)$$

For densities/mass functions:

$$P(x, y) = P(x)P(y) \qquad \text{or} \qquad p(x, y) = p(x)p(y)$$

Dependent means *not independent*.

Intuitively

X and Y are dependent if knowing the outcome of X provides any information about the outcome of Y.

More precisely:

- If someone draws (X, Y) simultanuously, and only discloses $X = x$ to you, does that change your mind about the distribution of Y? (If so: Dependence.)
- Once X is given, the distribution of Y is the conditional $\mathcal{L}(Y|X = x)$.
- If that is still $\mathcal{L}(Y)$, as before X was drawn, the two are independent. If $\mathcal{L}(Y|X = x) \neq \mathcal{L}(Y)$, they are dependent.

CONDITIONAL INDEPENDENCE

Definition

Given random variables X, Y, Z, we say that X is **conditionally independent of** Y **given** Z if

$$\mathcal{L}(X, Y|Z = z) = \mathcal{L}(X|Z = z)\mathcal{L}(Y|Z = z) \ .$$

That is equivalent to

$$\mathcal{L}(X|Y = y, Z = z) = \mathcal{L}(X|Z = z) \ .$$

Notation

$$X \perp\!\!\!\perp_Z Y$$

Intuitively

X and Y are dependent given $Z = z$ if, although Z is known, knowing the outcome of X provides additional information about the outcome of Y.

EXAMPLE: MARKOV CHAINS (OF ORDER 1)

Consider a Markov chain (X_0, X_1, X_2):

$$X_0 = x_0, X_1 = x_1, X_2 = ?$$

X_2 is conditionally independent of X_0 given X_1:

$$\mathcal{L}(X_2 | X_1 = x_1, X_0 = x_0) = \mathcal{L}(X_2 | X_1 = x_1)$$

However: If we do not condition on X_1, then X_2 need *not* be independent of X_0:

$$\mathcal{L}(X_2 | X_0 = x_0) \neq \mathcal{L}(X_2)$$

In general

For a Markov chain of order 1,

$$X_n \perp\!\!\!\perp_{X_{n-1}} X_{n-2}, \ldots, X_0$$

but not

$$X_n \perp\!\!\!\perp X_{n-2}, \ldots, X_0$$

GRAPHICAL MODEL NOTATION

Factorizing a joint distribution

The joint probability of random variables $X_1, \ldots, X_n$ can always be factorized as

$$\mathcal{L}(X_1, \ldots, X_n) = \mathcal{L}(X_n | X_1, \ldots, X_{n-1}) \mathcal{L}(X_{n-1} | X_1, \ldots, X_{n-2}) \cdots \mathcal{L}(X_1) \ .$$

Note that we can re-arrange the variables in any order.

If there are conditional independencies, we can remove some variables from the conditionals:

$$\mathcal{L}(X_1, \ldots, X_n) = \mathcal{L}(X_n | \mathcal{X}_n) \mathcal{L}(X_{n-1} | \mathcal{X}_{n-1}) \cdots \mathcal{L}(X_1) \ ,$$

where $\mathcal{X}_i$ is the subset of $X_1, \ldots, X_n$ on which X_i depends.

Definition

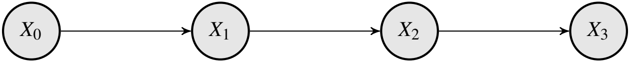
Let $X_1, \ldots, X_n$ be random variables. A **(directed) graphical model** represents a factorization of joint distribution $\mathcal{L}(X_1, \ldots, X_n)$ as follows:

- Factorize $\mathcal{L}(X_1, \ldots, X_n)$.
- Add one vertex for each variable X_i.
- For each variable X_i, add and edge from each variable $X_j \in \mathcal{X}_i$ to X_i.

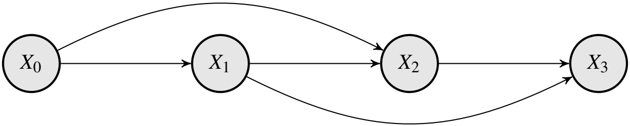
That is: An edge $X_j \rightarrow X_i$ is added if $\mathcal{L}(X_1, \ldots, X_n)$ contains the factor $\mathcal{L}(X_i | X_j)$.

EXAMPLE: MARKOV CHAINS

Markov chain of order $r = 1$



Markov chain of order $r = 2$



Note these graphs are graphical models, not Markov chain transition diagrams as those on the Markov chain slides. Here, each node is a random variable. In transition diagrams, each node represents a possible value of these variables.

Graphical Model Notation

Lack of uniqueness

The factorization is usually not unique, since e.g.

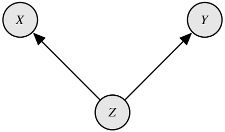
$$\mathcal{L}(X, Y) = \mathcal{L}(X|Y)\mathcal{L}(Y) = \mathcal{L}(Y|X)\mathcal{L}(X) \; .$$

That means the direction of edges is not generally determined.

Remark

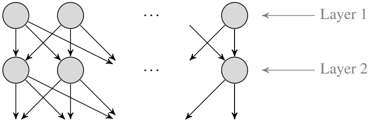
- If we use a graphical model to *define* a model or visualize a model, we decide on the direction of the edges.
- Estimating the direction of edges from data is a very difficult (and very important) problem. This is one of the main subjects of a research field called **causal inference** or **causality**.

A simple example



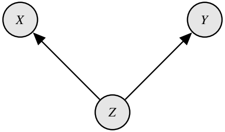
$$X \perp\!\!\!\perp_Z Y$$

An example with layers



All variables in the $(k+1)$st layer are conditionally independent given the variables in the kth layer.

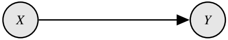
WORDS OF CAUTION I



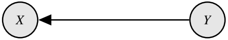
$$X \perp\!\!\!\perp_Z Y$$

Important

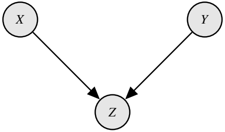
- *X* and *Y* are *not* independent, independence holds only conditionally on *Z*.
- In other words: If we do not observe *Z*, *X* and *Y* are dependent, and we have to change the graph:



or



WORDS OF CAUTION II



Conditioning on Z makes X and Y dependent.

Example

- Suppose we start with two indepedent normal variables X and Y.
- $Z = X + Y$.

If we know Z, and someone reveals the value of Y to us, we know everything about X.

This effect is known as *explaining away*. We will revisit it later.

HIDDEN MARKOV MODELS

OVERVIEW

Motivation

We have already used Markov models to model sequential data. Various important types of sequence data (speech etc) have long-range dependencies that a Markov model does not capture well.

Hidden Markov model

- A hidden Markov model is a latent variable model in which a sequence of latent (or "hidden") variables is generated by a Markov chain.
- These models can generate sequences of *observations* with long-range dependencies, but the *explanatory* variables (the latent variables) are Markovian.
- It turns out that this is a useful way to model dependence for a variety of important problems, including speech recognition, handwriting recognition, and parsing problems in genetics.

HIDDEN MARKOV MODELS

Definition

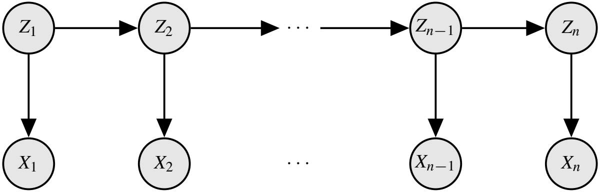
A (**discrete**) **hidden Markov model (HMM)** consists of:

- A stationary Markov chain $(Q_{\text{init}}, \mathbf{q})$ with states $\{1, \dots, K\}$, initial distribution Q_{init} and transition matrix $\mathbf{q}$.
- A (discrete) **emission distribution**, given by a conditional probability $P(x|z)$.

The model generates a sequence $X_1, X_2, \dots$ by:

1. Sampling a sequence $Z_1, Z_2, \dots$ from the Markov chain $(Q_{\text{init}}, \mathbf{q})$.
2. Sampling a sequence $X_1, X_2, \dots$ by independently sampling $X_i \sim P(\,.\,|Z_i)$.

In a **continuous HMM**, the variables X_i have continuous distributions, and $P(x|z)$ is substituted by a density $p(x|z)$. The Markov chain still has finite state space $[K]$.



NOTATION

We will see a lot of sequences, so we use the "programming" notation

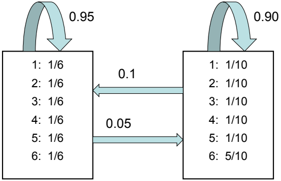
$$x_{1:n} := (x_1, \ldots, x_n)$$

Example: Dishonest Casino

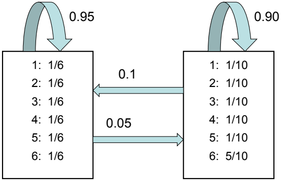
This example is used in many textbooks and is very simple, but it is useful to understand the conditional independence structure.

Problem

- We consider two dice (one fair, one loaded).
- At each roll, we either keep the current dice, or switch to the other one with a certain probability.
- A roll of the chosen dice is then observed.



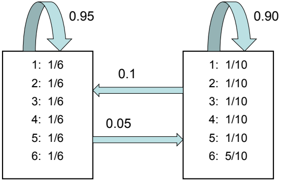
EXAMPLE: DISHONEST CASINO



HMM

- States: $Z_n \in \{\text{fair}, \text{loaded}\}$.
- Sample space: $\mathbf{X} = \{1, \dots, 6\}$.
- Transition matrix: $\mathbf{q} = \begin{pmatrix} 0.95 & 0.05 \\ 0.10 & 0.90 \end{pmatrix}$
- Emission probabilities:
  $P(x|z = \text{fair}) = (1/6, 1/6, 1/6, 1/6, 1/6, 1/6)$
  $P(x|z = \text{loaded}) = (1/10, 1/10, 1/10, 1/10, 1/10, 5/10)$

EXAMPLE: DISHONEST CASINO



Conditional independence

- Given the state (=which dice), the outcomes are independent.
- If we do not know the current state, observations are dependent!
- For example: If we observe a sequence of sixes, we are more likely to be in state "loaded" than "fair", which increases the probability of the next observation being a six.

HMM: Estimation Problems

Filtering problem

- **Given:** Model and observations, i.e. :
    1. Transition matrix $\mathbf{q}$ and emission distribution $P(\,.\, |z)$.
    2. Observed sequence $x_{1:N} = (x_1, \dots, x_N)$.
- **Estimate:** Probability of each hidden variable, i.e. $Q(Z_n = k|x_{1:n})$

Variant: **Smoothing problem**, in which we estimate $Q(Z_n = k|x_{1:N})$ instead.

Decoding problem

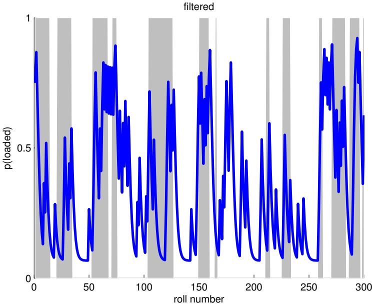
- **Given:** Model ($\mathbf{q}$ and $P(\,.\, |z)$) and observed sequence $x_{1:N}$.
- **Estimate:** Maximum likelihood estimates $\hat{z}_{1:N} = (\hat{z}_1, \dots, \hat{z}_N)$ of hidden states.

Learning problem

- **Given:** Observed sequence $x_{1:N}$.
- **Estimate:** Model (i.e. $\mathbf{q}$ and $P(\,.\, |z)$).

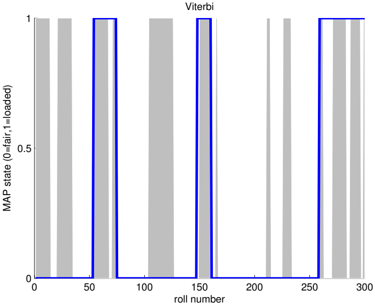
Examples

Before we look at the details, here are examples for the dishonest casino.



Filtering result.

Gray bars: Loaded dice used.
Blue: Probability $P(Z_n = \text{loaded}|x_{1:n})$



Decoding result.

Gray bars: Loaded dice used.
Blue: Most probable state Z_n.

PROBABILITIES OF HIDDEN STATES

The first estimation problem we consider is to estimate the probabilities $Q(z_n|x_{1:n})$.

Idea

We could use Bayes' equation (recall: $P(a|b) = \frac{P(b|a)P(a)}{P(b)}$) to write:

$$Q(k|x_n) = \frac{P(x_n|k)Q(Z_n = k)}{\sum_{k=1}^{K} P(x_n|k)Q(Z_n = k)} \; .$$

Since we know the Markov chain $(Q_{\text{init}}, \mathbf{q})$, we can compute Q, and the emission probabilities $P(x_n|k)$ are given.

Filtering

The drawback of the solution above is that it throws away all information about the past. We get a better estimate of Z_n by taking $x_1, \ldots, x_{n-1}$ into account. Reducing the uncertainty in Z_n using $x_1, \ldots, x_{n-1}$ is called **filtering**.

FILTERING

Filtering problem

Our task is to estimate the probabilities $Q(z_n|x_{1:n})$. Since the sequence has length n and each Z_i can take K possible values, this is a $N \times K$-matrix $\hat{Q}$, with entries

$$\hat{Q}_{nk} := Q(Z_n = k|x_{1:n}) \ .$$

Decomposition using Bayes' equation

We can use Bayes' equation (recall: $P(a|b) = \frac{P(b|a)P(a)}{P(b)}$) to write:

$$Q(z_n|x_{1:n}) = Q(z_n|x_n, x_{1:(n-1)}) = \underbrace{\frac{\overbrace{P(x_n|z_n, x_{1:(n-1)})}^{\text{This is the emission probability } P(x_n|z_n) \text{ (conditional independence!)}}\overbrace{Q(z_n|x_{1:(n-1)})}^{\text{This is the crucial term}}}{\sum_{z_n=1}^{K} P(x_n|z_n, x_{1:(n-1)})Q(z_n|x_{1:(n-1)})}}_{\text{Normalization}}$$

Reduction to previous step

The crucial idea is that we can use the results computed for step $n-1$ to compute those for step n:

$$Q(Z_n = k|x_{1:(n-1)}) = \sum_{l=1}^{K} \underbrace{Q(Z_n = k|Z_{n-1} = l)}_{= q_{lk} \text{ (transition matrix)}} \underbrace{Q(Z_{n-1} = l|x_{1:(n-1)})}_{= \hat{Q}_{(n-1)l}}$$

Summary

In short, we can compute the numerator in the Bayes equation as

$$a_{nk} := P(x_n|z_n) \sum_{l=1}^{K} q_{lk} \hat{Q}_{(n-1)l} \ .$$

The normalization term is

$$\sum_{z_n=1}^{K} \Big(P(x_n|z_n) \sum_{l=1}^{K} q_{lk} \hat{Q}_{(n-1)l} \Big) = \sum_{j=1}^{K} a_{nj} \ .$$

Solution to the filtering problem: The forward algorithm

Given is a sequence $(x_1, \dots, x_N)$.

For $n = 1, \dots, N$, compute

$$a_{nk} := P(x_n|z_n) \sum_{l=1}^{K} q_{lk} \hat{Q}_{(n-1)l} \;,$$

and

$$\hat{Q}_{nk} = \frac{a_{nk}}{\sum_{j=1}^{K} a_{nj}} \;.$$

This method is called the **forward algorithm**.

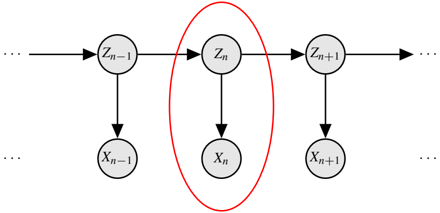
HMMs and Mixture Models

Parametric emission model

We usually define the emission probabilities $P(x_n|z_n)$ using a parametric model $P(x|\theta)$ (e.g. a multinomial or Gaussian model). Then

$$P(x_n|Z_n = k) := P(x_n|\theta_k) \ ,$$

i.e. the emission distribution of each state k is defined by a parameter value θ_k.



Relation to mixture models

If we just consider a *single* pair (Z_n, X_n), this defines a finite mixture with K clusters:

$$\pi(x_n) = \sum_{k=1}^{K} c_k P(x_n|\theta_k) = \sum_{k=1}^{K} Q(Z_n = k) P(x_n|\theta_k)$$

EM FOR HMMS

Recall: EM for mixtures

E-step	M-step
Soft assignments $\mathbb{E}[M_{ik}] = \Pr(m_i = k)$	cluster weights c_k component parameters θ_k

HMM case

- For mixtures, $\Pr\{m_i = k\} = c_k$. In HMMs, the analogous probability $\Pr\{Z_n = k\}$ is determined by the transition probabilities.
- The analogue of the soft assignments a_{ik} computed for mixtures are state probabilities

$$b_{nk} = Q(Z_n = k|\theta, x_{1:n}) \ .$$

- Additionally, we have to estimate the transition matrix $\mathbf{q}$ of the Markov chain.

EM for HMMs

E-step	M-step
Transition probabilities q_{kj} State probabilities b_{nk}	component parameters θ_k

EM for HMMs

M-step

The M-step works exactly as for mixture models. E.g. for Gaussian emission distributions with parameters μ_k and σ_k^2,

State probabilities substituted for assignment probabilities

$$\mu_k = \frac{\sum_{n=1}^{N} b_{nk} x_n}{\sum_{n=1}^{N} b_{nk}} \qquad \text{and} \qquad \sigma_k^2 = \frac{\sum_{n=1}^{N} b_{nk}(x_n - \mu_k)^2}{\sum_{n=1}^{N} b_{nk}}$$

E-step

- Computing the state probabilities is a filtering problem:

  $$b_{nk}^{\text{new}} = Q(Z_n = k | \theta^{\text{old}}, x_{1:n}) \; .$$

  The forward algorithm assumes the emission probabilities are known, so we use the emission parameters θ^{old} computed during the previous M-step.

- Estimating the transition probabilities is essentially a filtering-type problem for *pairs* of states and can also be solved recursively, but we will skip the details since the equations are quite lengthy.

APPLICATION: SPEECH RECOGNITION

Problem

Given speech in form of a sound signal, determine the words that have been spoken.

Method

- Words are broken down into small sound units (called *phonemes*). The states in the HMM represent phonemes.
- The incoming sound signal is transformed into a sequence of vectors (feature extraction). Each vector x_n is indexed by a time step n.
- The sequence $x_{1:N}$ of feature vectors is the observed data in the HMM.

PHONEME MODELS

Phoneme

A **phoneme** is defined as the smallest unit of sound in a language that distinguishes between distinct meanings. English uses about 50 phonemes.

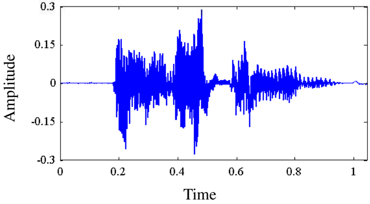
Example

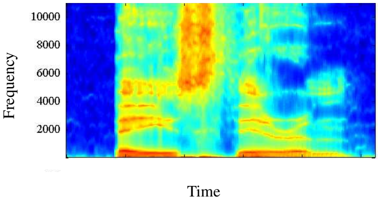
Zero	Z IH R OW	Six	S IH K S
One	W AH N	Seven	S EH V AX N
Two	T UW	Eight	EY T
Three	TH R IY	Nine	N AY N
Four	F OW R	Oh	OW
Five	F AY V		

Subphonemes

Phonemes can be further broken down into subphonemes. The standard in speech processing is to represent a phoneme by three subphonemes ("triphons").

PREPROCESSING SPEECH

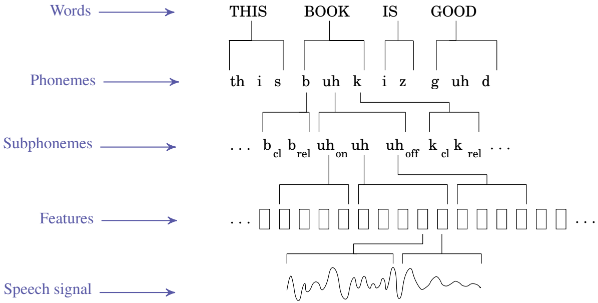




Feature extraction

- A speech signal is measured as amplitude over time.
- The signal is typically transformed into various types of features, including (windowed) Fourier- or cosine-transforms and so-called "cepstral features".
- Each of these transforms is a scalar function of time. All function values for the different transforms at time *t* are collected in a vector, which is the feature vector (at time *t*).

LAYERS IN PHONEME MODELS



HMM speech recognition

- Training: The HMM parameters (emission parameters and transition probabilities) are estimated from data, often using both supervised and unsupervised techniques.
- Recognition: Given a language signal (= observation sequence $x_{1:N}$, estimate the corresponding sequence of subphonemes (= states $z_{1:N}$). This is a decoding problem.

Speaker Adaptation

Factory model

Training requires a lot of data; software is typically shipped with a model trained on a large corpus (i.e. the HMM parameters are set to "factory settings").

The adaptation problem

- The factory model represents an average speaker. Recognition rates can be improved drastically by adapting to the specific speaker using the software.
- Before using the software, the user is presented with a few sentences and asked to read them out, which provides labelled training data.

Speaker adaptation

- Transition probabilities are properties of the language. Differences between speakers (pronounciation) are reflected by the emission parameters θ_k.
- Emission probabilities in speech are typically multi-dimensional Gaussians, so we have to adapt means and covariance matrices.
- The arguably most widely used method is **maximum likelihood linear regression** (**MLLR**), which uses a regression technique to make small changes to the covariance matrices.

FURTHER READING

More details on HMMs

If you feel enthusiastic, the following books provide more background:

- David Barber’s "Bayesian reasoning and machine learning" (available online).
- Chris Bishop’s "Pattern recognition and machine learning".
- Many books on speech, e.g. Rabiner’s classic "Fundamentals of speech recognition".

HTK

If you would like to try out speech recognition software, have a look at the HTK (**H**MM **T**ool**k**it) package, which is the de-facto standard in speech research. HTK implements both HMMs for recognition and routines for feature extraction.

BAYESIAN MIXTURE MODELS

OVERVIEW

The concept of a Bayesian mixture will come up a few times in this class, so we will briefly review it on the next few slides.

Idea

- Recall that the defining idea of a Bayesian model is to treat the model parameters as random variables.
- That requires specifying a distribution for those parameters, known as the *prior distribution*.
- Instead of asking for an value of the parameters estimated from data (a *point estimate*), we determine their conditional distribution given the data. This distribution is called the *posterior distribution*.
- A Bayesian mixture model is a finite mixture model whose model parameters are treated as random.

Inference: Sampling

These models are examples of models in which the exact posterior is intractable. Inference uses Markov chain Monte Carlo sampling, which will be our main topic for the last two lectures.

Parameters of Bayesian Mixtures

Recall: Finite mixture models

$$\pi(x) = \sum_{k=1}^{K} c_k p(x|\theta_k)$$

The model parameters are the component parameters θ_k and the weights c_k. For a Bayesian version of the model, we hence have to generate random component parameters $\Theta_1, \dots, \Theta_K$ and random weights $C_1, \dots, C_K$.

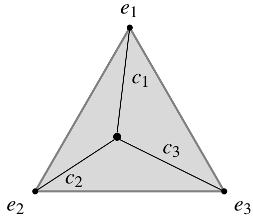
More precisely

- The mixture components $p(x|\theta)$ are an exponential family model (as discussed in SML).
- Which prior we choose for Θ depends on the choice of p.
- The prior of the vector $(C_1, \dots, C_K)$ is always chose as a *Dirichlet distribution*.

The Dirichlet distribution

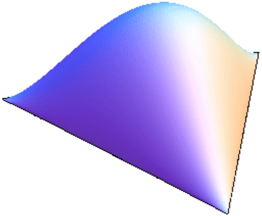
The Dirichlet distribution, for some specified number $K \in \mathbb{N}$, generates a random vector $(C_1, \dots, C_K)$ with the property that $C_k > 0$ for all $k = 1, \dots, K$ and $\sum_{k \leq K} C_k = 1$. In other words, $(C_1, \dots, C_K)$ is a (randomly generated) probability distribution on K categories.

THE DIRICHLET DISTRIBUTION



Recall: Probability simplex

The set of all probability distributions on K events is the *simplex* $\Delta_K := \{(c_1, \ldots, c_k) \in \mathbb{R}^K \mid c_k \geq 0 \text{ and } \sum_k c_K = 1\}$.



Dirichlet distribution

The **Dirichlet distribution** is the distribution on Δ_K with density

$$q_{\text{Dirichlet}}(c_{1:K}|\alpha, g_{1:K}) := \frac{1}{K(\alpha, g_{1:K})} \exp\Big(\sum_{k=1}^{K}(\alpha g_k - 1)\log(c_k)\Big)$$

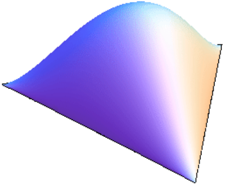
Parameters:

- $g_{1:K} \in \Delta_K$: Mean parameter, i.e. $\mathbb{E}[c_{1:K}] = g_{1:K}$.
- $\alpha \in \mathbb{R}_+$: Concentration.
  Larger $\alpha \rightarrow$ sharper concentration around $g_{1:K}$.

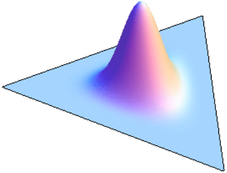
The Dirichlet Distribution

In all plots, $g_{1:K} = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$. Light colors = large density values.

Density plots

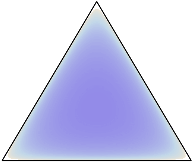


$\alpha = 1.8$

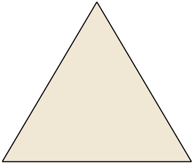


$\alpha = 10$

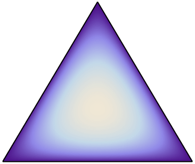
As heat maps



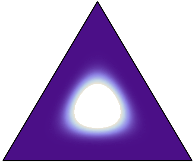
$\alpha = 0.8$
Large density values
at extreme points



$\alpha = 1$
Uniform distribution
on Δ_K



$\alpha = 1.8$
Density peaks
around its mean



$\alpha = 10$
Peak sharpens
with increasing α

Multinomial-Dirichlet Model

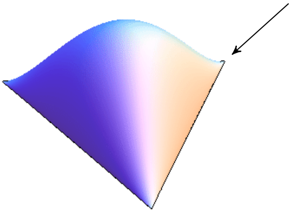
- Suppose we observe data from a multinomial distribution on K categories. The multinomial is parametrized by a finite probability distribution on K events. We generate this distribution using a Dirichlet (with hyperparameters α and $g_{1:K}$). What is the posterior?
- If we observe h_k counts in category k, the posterior is

$$\Pi(c_{1:K}|h_1,\dots,h_k) = q_{\text{Dirichlet}}(c_{1:K}|\alpha+n,(\alpha g_1+h_1,\dots,\alpha g_K+h_K))$$

  where $n = \sum_k h_k$ is the total number of observations.
- The posterior distribution of a Dirichlet prior combined with a multinomial observation model is again a Dirichlet distribution.

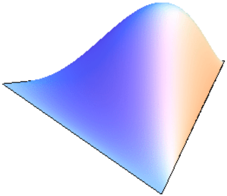
Illustration: One observation

Suppose $K = 3$ and we obtain a single observation in category 3.



This extreme point correponds to $k = 3$.

Prior: Mean at the center.



Posterior: Shifted mean, increased concentration.

BAYESIAN MIXTURE MODELS

Definition

A model of the form

$$\pi(x) = \sum_{k=1}^{K} C_k p(x|\Theta_k)$$

is called a **Bayesian mixture model** if $p(x|\theta)$ is an exponential family model and

- $\Theta_1, \ldots, \Theta_K \sim_{\text{iid}} q$, where q is a prior we have chosen for Θ.
- $(C_1, \ldots, C_K)$ is sampled from a K-dimensional Dirichlet distribution.

Bayesian Mixture: Inference

Posterior distribution

The posterior of a BMM under observations $x_1, \ldots, x_n$ is (up to normalization):

$$\Pi(c_{1:K}, \theta_{1:K} | x_{1:n}) \propto \prod_{i=1}^{n} \Big(\sum_{k=1}^{K} c_k p(x_i | \theta_k) \Big) \Big(\prod_{k=1}^{K} q(\theta_k) \Big) q_{\text{Dirichlet}}(c_{1:K})$$

The posterior is analytically intractable

- Thanks to conjugacy, we *can* evaluate each term of the posterior.
- However: Due to the $\prod_{i=1}^{n} \Big(\sum_{k=1}^{K} \ldots \Big)$ bit, the posterior has K^n terms!
- Even for 10 clusters and 100 observations, that is impossible to compute.

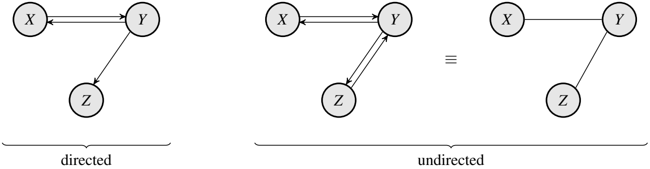
Solution

The posterior can be sampled with a very simple MCMC sampler (which looks strikingly similar to an EM algorithm, and will be discussed later).

Markov Random Fields

UNDIRECTED GRAPHICAL MODEL

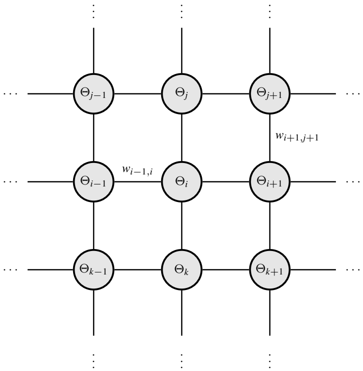
- A graphical model is **undirected** when its dependency graph is undirected; equivalently, if each edge in the graph is either absent, or present in both directions.



- An undirected graphical model is more commonly known as a **Markov random field**.
- Markov random fields are special cases of (directed) graphical models, but have distinct properties. We treat them separately.
- We will consider the undirected case first.

OVERVIEW

We start with an undirected graph:



A random variable Θ_i is associated with each vertex. Two random variables interact if they are neighbors in the graph.

NEIGHBORHOOD GRAPH

- We define a **neighborhood graph**, which is a weighted, undirected graph:

$$\mathcal{N} = (\overset{\text{vertex set}}{\downarrow \atop V_{\mathcal{N}}}, \overset{\text{set of edge weights}}{\downarrow \atop W_{\mathcal{N}}})$$

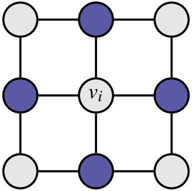
  The vertices $v_i \in V_{\mathcal{N}}$ are often referred to as **sites**.
- The edge weights are scalars $w_{ij} \in \mathbb{R}$. Since the graph is undirected, the weights are symmetric ($w_{ij} = w_{ji}$).
- An edge weight $w_{ij} = 0$ means "no edge between v_i and v_j".

Neighborhoods

The set of all neighbors of v_j in the graph,

$$\partial\,(i) := \{\, j \,|\, w_{ij} \neq 0\}$$

is called the **neighborhood** of v_i.



purple = $\partial\;(i)$

Markov Random Fields

Given a neighborhood graph $\mathcal{N}$, associate with each site $v_i \in V_{\mathcal{N}}$ a RV Θ_i.

The Markov property

We say that the joint distribution P of $(\Theta_1, \dots, \Theta_n)$ satisfies the **Markov property** with respect to $\mathcal{N}$ if

$$\mathcal{L}(\Theta_i|\Theta_j,\ j \neq i) = \mathcal{L}(\Theta_i|\Theta_j,\ j \in \partial(i))\ .$$

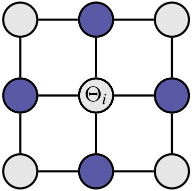
The set $\{\Theta_j, j \in \partial(i)\}$ of random variables indexed by neighbors of v_i is called the **Markov blanket** of Θ_i.

In words

The Markov property says that each Θ_i is conditionally independent of the remaining variables given its Markov blanket.

Definition

A distribution $\mathcal{L}(\Theta_1, \dots, \Theta_n)$ which satisfies the Markov property for a given graph $\mathcal{N}$ is called a **Markov random field**.



Markov blanket of Θ_i

Energy Functions

Probabilities and energies

A (strictly positive) density $p(x)$ can always be written in the form

$$p(x) = \frac{1}{Z}\exp(-H(x)) \qquad \text{where} \qquad H : \mathbf{X} \to \mathbb{R}_+$$

and Z is a normalization constant.

The function H is called an **energy function**, or **cost function**, or a **potential**.

MRF energy

In particular, we can write a MRF density for RVs $\Theta_{1:n}$ as

$$p(\theta_1, \dots, \theta_n) = \frac{1}{Z}\exp(-H(\theta_1, \dots, \theta_n))$$

The Potts Model

Definition

Suppose $\mathcal{N} = (V_{\mathcal{N}}, W_{\mathcal{N}})$ a neighborhood graph with n vertices and $\beta > 0$ a constant. Then

$$p(\theta_{1:n}) := \frac{1}{Z(\beta, W_{\mathcal{N}})} \exp\Big(\beta \sum_{i,j} w_{ij} \mathbb{I}\{\theta_i = \theta_j\}\Big)$$

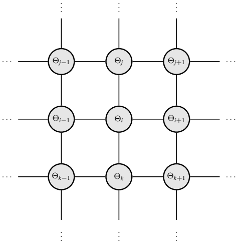
defines a joint distribution of n random variables $\Theta_1, \ldots, \Theta_n$. This distribution is called the **Potts model**.

Interpretation

- If $w_{ij} > 0$: The overall probability *increases* if $\Theta_i = \Theta_j$.
- If $w_{ij} < 0$: The overall probability *decreases* if $\Theta_i = \Theta_j$.
- If $w_{ij} = 0$: No interaction between Θ_i and Θ_j.

Positive weights encourage *smoothness*.

EXAMPLE



Ising model

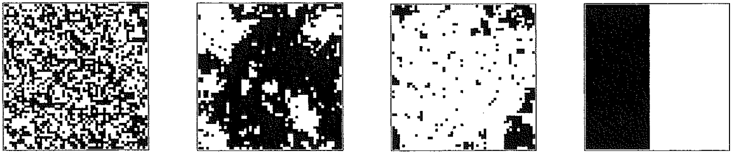
The simplest choice is $w_{ij} = 1$ if (i, j) is an edge.

$$p(\theta_{1:n}) = \frac{1}{Z(\beta)} \exp\Big(\sum_{(i,j) \text{ is an edge}} \beta \mathbb{I}\{\theta_i = \theta_j\} \Big)$$

If $\mathcal{N}$ is a d-dim. grid, this model is called the **Ising model**.

Example

Samples from an Ising model on a 56 × 56 grid graph.



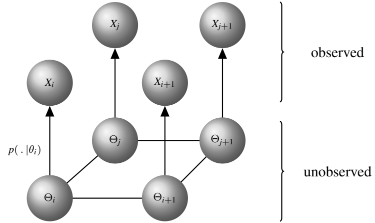
Increasing $\beta \quad \longrightarrow$

MRFs as Smoothness Priors

We consider a spatial problem with observations X_i. Each i is a location on a grid.

Spatial model

Suppose we model each X_i by a distribution $\mathcal{L}(X|\Theta_i)$, i.e. each location i has its own parameter variable Θ_i. This model is Bayesian (the parameter is a random variable). We use an MRF as prior distribution.



We can think of $\mathcal{L}(X|\Theta_i)$ as an emission probability, similar to an HMM.

Spatial smoothing

- We can define the joint distribution $(\Theta_1, \ldots, \Theta_n)$ as a MRF on the grid graph.
- For positive weights, the MRF will encourage the model to explain neighbors X_i and X_j by the same parameter value. → Spatial smoothing.

Example: Segmentation of Noisy Images

Mixture model

- A BMM can be used for image segmentation.
- The BMM prior on the component parameters is a natural conjugate prior $q(\theta)$.
- In the spatial setting, we index the parameter of each X_i separately as θ_i. For K mixture components, $\theta_{1:n}$ contains only K different values.
- The joint BMM prior on $\theta_{1:n}$ is

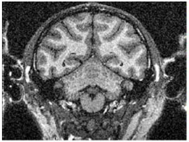
$$q_{\text{BMM}}(\theta_{1:n}) = \prod_{i=1}^{n} q(\theta_i) \ .$$

Smoothing term

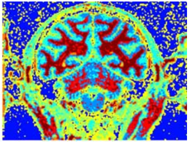
We multiply the BMM prior $q_{\text{BMM}}(\theta)$ with an MRF prior

$$q_{\text{MRF}}(\theta_{1:n}) = \frac{1}{Z(\beta)} \exp\Big(\beta \sum_{w_{ij} \neq 0} \mathbb{I}\{\theta_i = \theta_j\}\Big)$$

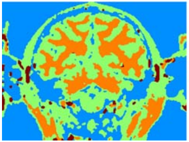
This encourages spatial smoothnes of the segmentation.



Input image.



Segmentation w/o smoothing.



Segmentation with MRF smoothing.

SAMPLING AND INFERENCE

MRFs pose two main computational problems.

Problem 1: Sampling

Generate samples from the joint distribution of $(\Theta_1, \dots, \Theta_n)$.

Problem 2: Inference

If the MRF is used as a prior, we have to compute or approximate the posterior distribution.

Solution

- MRF distributions on grids are *not* analytically tractable. The only known exception is the Ising model in 1 dimension.
- Both sampling and inference are based on Markov chain sampling algorithms.

SAMPLING ALGORITHMS

SAMPLING ALGORITHMS

In general

- A **sampling algorithm** is an algorithm that outputs samples $X_1, X_2, \ldots$ from a given distribution P or density p.
- Sampling algorithms can for example be used to approximate expectations:

$$\mathbb{E}_p[f(X)] \approx \frac{1}{n} \sum_{i=1}^{n} f(X_i)$$

Inference in Bayesian models

Suppose we work with a Bayesian model whose posterior $\hat{Q}_n := \mathcal{L}(\Theta | X_{1:n})$ cannot be computed analytically.

- We will see that it can still be possible to *sample* from $\hat{Q}_n$.
- Doing so, we obtain samples $\Theta_1, \Theta_2, \ldots$ distributed according to $\hat{Q}_n$.
- This reduces posterior estimation to a density estimation problem (i.e. estimate $\hat{Q}_n$ from $\Theta_1, \Theta_2, \ldots$).

PREDICTIVE DISTRIBUTIONS

Posterior expectations

If we are only interested in some statistic of the posterior of the form $\mathbb{E}_{\hat{Q}_n}[f(\Theta)]$ (e.g. the posterior mean), we can again approximate by

$$\mathbb{E}_{\hat{Q}_n}[f(\Theta)] \approx \frac{1}{m}\sum_{i=1}^{m} f(\Theta_i) \ .$$

Example: Predictive distribution

The **posterior predictive distribution** is our best guess of what the next data point x_{n+1} looks like, given the posterior under previous observations. In terms of densities:

$$p(x_{n+1}|x_{1:n}) := \int_{\mathbf{T}} p(x_{n+1}|\theta)\hat{Q}_n(d\theta|X_{1:n} = x_{1:n}) \ .$$

This is one of the key quantities of interest in Bayesian statistics.

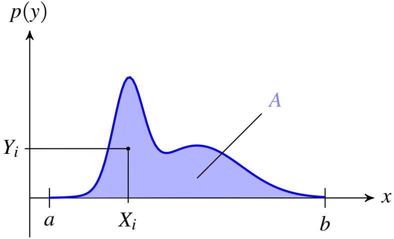
Computation from samples

The predictive is a posterior expectation, and can be approximated as a sample average:

$$p(x_{n+1}|x_{1:n}) = \mathbb{E}_{\hat{Q}_n}[p(x_{n+1}|\Theta)] \approx \frac{1}{m}\sum_{i=1}^{m} p(x_{n+1}|\Theta_i)$$

Basic Sampling: Area Under Curve

Say we are interested in a probability density p on the interval $[a, b]$.



Key observation

Suppose we can define a uniform distribution U_A on the blue area A under the curve. If we sample

$$(X_1, Y_1), (X_2, Y_2), \ldots \sim_{\text{iid}} U_A$$

and discard the vertical coordinates Y_i, **the X_i are distributed according to** p,

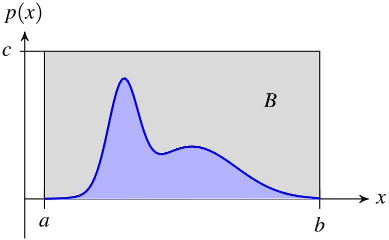
$$X_1, X_2, \ldots \sim_{\text{iid}} p \ .$$

Problem: Defining a uniform distribution is easy on a rectangular area, but difficult on an arbitrarily shaped one.

Rejection Sampling on the Interval

Solution: Rejection sampling

We can enclose p in box, and sample uniformly from the box B.

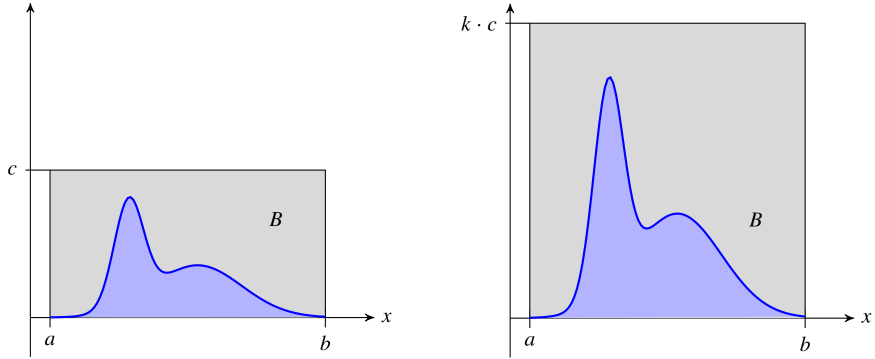


- We can sample (X_i, Y_i) uniformly on B by sampling
$$X_i \sim \text{Uniform}[a, b] \qquad \text{and} \qquad Y_i \sim \text{Uniform}[0, c]\ .$$
- If $(X_i, Y_i) \in A$, keep the sample.
That is: If $Y_i \leq p(X_i)$.
- Otherwise: Discard it ("reject" it).

Result: The remaining (non-rejected) samples are uniformly distributed on A.

SCALING

This strategy still works if we scale the vertically by some constant $k > 0$.



We simply draw $Y_i \sim \text{Uniform}[0, kc]$ instead of $Y_i \sim \text{Uniform}[0, c]$.

Consequence

For sampling, it is sufficient if p is known only up to normalization (only the shape of p is known).

DISTRIBUTIONS KNOWN UP TO SCALING

Sampling methods usually assume that we can evaluate the target distribution p up to a constant. That is:

$$p(x) \quad = \quad \frac{1}{\tilde{Z}}\,\tilde{p}(x)\ ,$$

and we can compute $\tilde{p}(x)$ for any given x, but we do not know $\tilde{Z}$.

We have to pause for a moment and convince ourselves that there are useful examples where this assumption holds.

Example 1: Simple posterior

For an arbitrary posterior computed with Bayes' theorem, we could write

$$\Pi(\theta|x_{1:n}) = \frac{\prod_{i=1}^{n} p(x_i|\theta)q(\theta)}{\tilde{Z}} \qquad \text{with} \qquad \tilde{Z} = \int_{\mathbf{T}} \prod_{i=1}^{n} p(x_i|\theta)q(\theta)d\theta\ .$$

Provided that we can compute the numerator, we can sample without computing the normalization integral $\tilde{Z}$.

Example 2: Bayesian Mixture Model

Recall that the posterior of the BMM is (up to normalization):

$$\hat{q}_n(c_{1:K}, \theta_{1:K}|x_{1:n}) \propto \prod_{i=1}^{n}\Big(\sum_{k=1}^{K} c_k p(x_i|\theta_k)\Big)\Big(\prod_{k=1}^{K} q(\theta_k)\Big) q_{\text{Dirichlet}}(c_{1:K})$$

We already know that we can discard the normalization constant, but can we evaluate the non-normalized posterior $\tilde{q}_n$?

- The problem with computing $\tilde{q}_n$ (as a function of unknowns) is that the term $\prod_{i=1}^{n}\Big(\sum_{k=1}^{K} \dots\Big)$ blows up into K^n individual terms.
- If we *evaluate* $\tilde{q}_n$ for specific values of c, x and θ, $\sum_{k=1}^{K} c_k p(x_i|\theta_k)$ collapses to a single number for each x_i, and we just have to multiply those n numbers.

So: Computing $\tilde{q}_n$ as a formula in terms of unknowns is difficult; evaluating it for specific values of the arguments is easy.

Example 3: Markov random field

In a MRF, the normalization function is the real problem.

For example, recall the Ising model:

$$p(\theta_{1:n}) = \frac{1}{Z(\beta)} \exp\Big(\sum_{(i,j) \text{ is an edge}} \beta \mathbb{I}\{\theta_i = \theta_j\} \Big)$$

The normalization function is

$$Z(\beta) = \sum_{\theta_{1:n} \in \{0,1\}^n} \exp\Big(\sum_{(i,j) \text{ is an edge}} \beta \mathbb{I}\{\theta_i = \theta_j\} \Big)$$

and hence a sum over 2^n terms. The general Potts model is even more difficult.

On the other hand, evaluating

$$\tilde{p}(\theta_{1:n}) = \exp\Big(\sum_{(i,j) \text{ is an edge}} \beta \mathbb{I}\{\theta_i = \theta_j\} \Big)$$

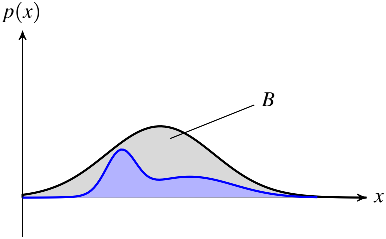
for a given configuration $\theta_{1:n}$ is straightforward.

Rejection Sampling on $\mathbb{R}^d$

If we are not on the interval, sampling uniformly from an enclosing box is not possible (since there is no uniform distribution on all of $\mathbb{R}$ or $\mathbb{R}^d$).

Solution: Proposal density

Instead of a box, we use *another distribution* r to enclose p:

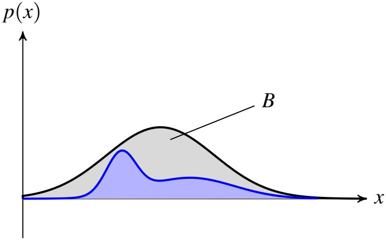


To generate B under r, we apply similar logic as before backwards:

- Sample $X_i \sim r$.
- Sample $Y_i | X_i \sim \text{Uniform}[0, r(X_i)]$.

r is always a simple distribution which we can sample and evaluate.

REJECTION SAMPLING ON $\mathbb{R}^d$



- Choose a simple distribution r from which we know how to sample.
- Scale $\tilde{p}$ such that $\tilde{p}(x) < r(x)$ everywhere.
- Sampling: For $i = 1, 2, \ldots,$:
  1. Sample $X_i \sim r$.
  2. Sample $Y_i | X_i \sim \text{Uniform}[0, r(X_i)]$.
  3. If $Y_i < \tilde{p}(X_i)$, keep X_i.
  4. Else, discard X_i and start again at (1).
- The surviving samples $X_1, X_2, \ldots$ are distributed according to p.

Factorization Perspective

The rejection step can be interpreted in terms of probabilities and densities.

Factorization

We factorize the target distribution or density p as

distribution from which we know how to sample

$$p(x) \quad = \quad r(x) \cdot A(x)$$

probability function we can evaluate *once a specific value of x is given*

Sampling from the factorization

$$X \quad = \quad \begin{cases} X' & \text{if } Z = 1 \\ \text{(discard)} & \text{if } Z = 0 \end{cases}$$

where $\quad X' \sim r \quad$ and $\quad Z|X' \sim \text{Bernoulli}(A(X'))$

Sampling Bernoulli variables with uniform variables

$$Z|X' \sim \text{Bernoulli}(A(X')) \qquad \Leftrightarrow \qquad Z = \mathbb{I}\{U < A(X')\} \quad \text{where} \quad U \sim \text{Uniform}[0,1]\ .$$

Not examinable.

INDEPENDENCE

If we draw proposal samples X_i i.i.d. from r, the resulting sequence of accepted samples produced by rejection sampling is again i.i.d. with distribution p. Hence:

Rejection samplers produce i.i.d. sequences of samples.

Important consequence

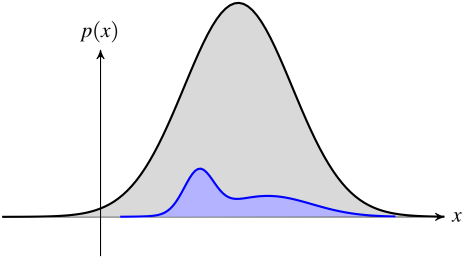
If samples $X_1, X_2, \dots$ are drawn by a rejection sampler, the sample average

$$\frac{1}{m}\sum_{i=1}^{m} f(X_i)$$

(for some function f) is an unbiased estimate of the expectation $\mathbb{E}_p[f(X)]$.

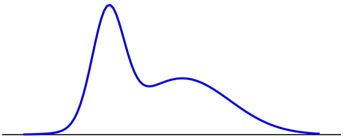
EFFICIENCY

The fraction of accepted samples is the ratio $\frac{|A|}{|B|}$ of the areas under the curves $\tilde{p}$ and r.

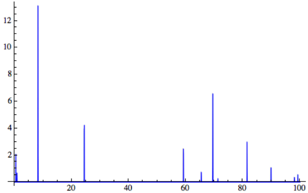


If r is not a reasonably close approximation of p, we will end up rejecting a lot of proposal samples.

AN IMPORTANT BIT OF IMPRECISE INTUITION



Example figures for sampling methods tend to look like this.



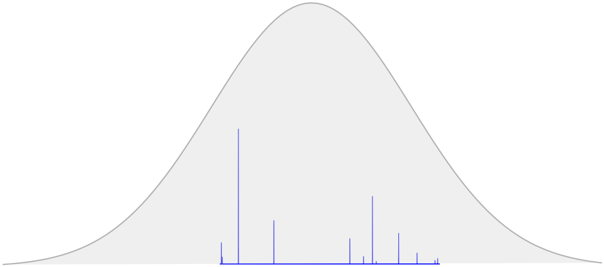
A high-dimensional distribution of correlated RVs will look rather more like this.

Sampling is usually used in multiple dimensions. Reason, roughly speaking:

- Intractable posterior distributions arise when there are several *interacting* random variables. The interactions make the joint distribution complicated.
- In one-dimensional problems (1 RV), we can usually compute the posterior analytically.
- Independent multi-dimensional distributions factorize and reduce to one-dimensional case.

Warning: Avoid sampling if you can solve analytically.

WHY IS NOT EVERY SAMPLER A REJECTION SAMPLER?



We can easily end up in situations where we accept only one in 10^6 (or 10^{10}, or 10^{20},...) proposal samples. Especially in higher dimensions, we have to expect this to be not the exception but the rule.

Importance Sampling

The rejection problem can be fixed easily if we are only interested in approximating an expectation $\mathbb{E}_p[f(X)]$.

Simple case: We can evaluate p

Suppose p is the target density and q a proposal density. An expectation under p can be rewritten as

$$\mathbb{E}_p[f(X)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = \mathbb{E}_q\left[\frac{f(X)p(X)}{q(X)}\right]$$

Importance sampling

We can sample $X_1, X_2, \dots$ from q and approximate $\mathbb{E}_p[f(X)]$ as

$$\mathbb{E}_p[f(X)] \approx \frac{1}{m}\sum_{i=1}^{m} f(X_i)\frac{p(X_i)}{q(X_i)}$$

There is no rejection step; all samples are used.

This method is called **importance sampling**. The coefficients $\frac{p(X_i)}{q(X_i)}$ are called **importance weights**.

Importance Sampling

General case: We can only evaluate $\tilde{p}$

In the general case,

$$p = \frac{1}{Z_p}\tilde{p} \qquad \text{and} \qquad q = \frac{1}{Z_q}\tilde{q}\ ,$$

and Z_p (and possibly Z_q) are unknown. We can write $\frac{Z_p}{Z_q}$ as

$$\frac{Z_p}{Z_q} = \frac{\int \tilde{p}(x)dx}{Z_q} = \frac{\int \tilde{p}(x)\frac{q(x)}{q(x)}dx}{Z_q} = \int \tilde{p}(x)\frac{q(x)}{Z_q \cdot q(x)}dx = \mathbb{E}_q\left[\frac{\tilde{p}(X)}{\tilde{q}(X)}\right]$$

Approximating the constants

The fraction $\frac{Z_p}{Z_q}$ can be approximated using samples $x_{1:m}$ from q:

$$\frac{Z_p}{Z_q} \quad = \quad \mathbb{E}_q\left[\frac{\tilde{p}(X)}{\tilde{q}(X)}\right] \quad \approx \quad \frac{1}{m}\sum_{i=1}^{m}\frac{\tilde{p}(X_i)}{\tilde{q}(X_i)}$$

Approximating $\mathbb{E}_p[f(X)]$

$$\mathbb{E}_p[f(X)] \quad \approx \quad \frac{1}{m}\sum_{i=1}^{m} f(X_i)\frac{p(X_i)}{q(X_i)} \quad = \quad \frac{1}{m}\sum_{i=1}^{m} f(X_i)\frac{Z_q}{Z_p}\frac{\tilde{p}(X_i)}{\tilde{q}(X_i)} \quad = \quad \sum_{i=1}^{m}\frac{f(X_i)\frac{\tilde{p}(X_i)}{\tilde{q}(X_i)}}{\sum_{j=1}^{m}\frac{\tilde{p}(X_j)}{\tilde{q}(X_j)}}$$

IMPORTANCE SAMPLING IN GENERAL

Conditions

- Given are a target distribution p and a proposal distribution q.
- $p = \frac{1}{Z_p}\tilde{p}$ and $q = \frac{1}{Z_q}\tilde{q}$.
- We can evaluate $\tilde{p}$ and $\tilde{q}$, and we can sample q.
- The objective is to compute $\mathbb{E}_p[f(X)]$ for a given function f.

Algorithm

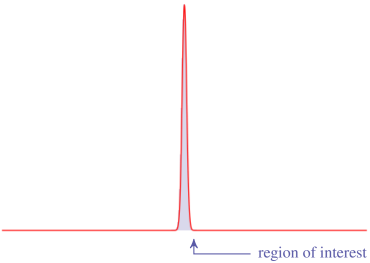
1. Sample $X_1, \dots, X_m$ from q.
2. Approximate $\mathbb{E}_p[f(X)]$ as

$$\mathbb{E}_p[f(X)] \approx \frac{\sum_{i=1}^m f(X_i)\frac{\tilde{p}(X_i)}{\tilde{q}(X_i)}}{\sum_{j=1}^m \frac{\tilde{p}(X_j)}{\tilde{q}(X_j)}}$$

Markov Chain Monte Carlo

MOTIVATION

Suppose we rejection-sample a distribution like this:



Once we have drawn a sample in the narrow region of interest, we would like to continue drawing samples within the same region. That is only possible if each sample *depends on the location of the previous sample*.

Proposals in rejection sampling are i.i.d. Hence, once we have found the region where p concentrates, we forget about it for the next sample.

MCMC: IDEA

Recall: Markov chain

- A sufficiently nice Markov chain (MC) has an invariant distribution P_{inv}.
- Once the MC has converged to P_{inv}, each sample X_i from the chain has marginal distribution P_{inv}.

Markov chain Monte Carlo

We want to sample from a distribution with density p. Suppose we can define a MC with invariant distribution $P_{\text{inv}} \equiv p$. If we sample $X_1, X_2, \ldots$ from the chain, then once it has converged, we obtain samples

$$X_i \sim p \ .$$

This sampling technique is called **Markov chain Monte Carlo** (**MCMC**).

Note: For a Markov chain, X_{i+1} can depend on X_i, so at least in principle, it is possible for an MCMC sampler to "remember" the previous step and remain in a high-probability location.

Continuous Markov Chain

The Markov chains we discussed so far had a finite state space $\mathbf{X}$. For MCMC, state space now has to be the domain of p, so we often need to work with continuous state spaces.

Continuous Markov chain

A continuous Markov chain is defined by an initial distribution P_{init} and conditional probability $t(y|x)$, the **transition probability** or **transition kernel**.

In the discrete case, $t(y = i|x = j)$ is the entry $\mathbf{p}_{ij}$ of the transition matrix $\mathbf{p}$.

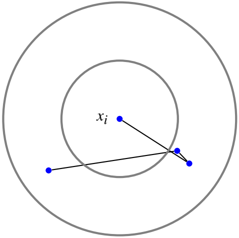
Example: A Markov chain on $\mathbb{R}^2$

We can define a very simple Markov chain by sampling

$$X_{i+1}|X_i = x_i \quad \sim \quad g(\,.\,|x_i, \sigma^2)$$

where $g(x|\mu, \sigma^2)$ is a spherical Gaussian with fixed variance. In other words, the transition distribution is

$$t(x_{i+1}|x_i) := g(x_{i+1}|x_i, \sigma^2)\ .$$



A Gaussian (gray contours) is placed around the current point x_i to sample X_{i+1}.

INVARIANT DISTRIBUTION

Recall: Finite case

- The invariant distribution P_{inv} is a distribution on the finite state space $\mathbf{X}$ of the MC (i.e. a vector of length $|\mathbf{X}|$).
- "Invariant" means that, if X_i is distributed according to P_{inv}, and we execute a step $X_{i+1} \sim t(\,.\,|x_i)$ of the chain, then X_{i+1} again has distribution P_{inv}.
- In terms of the transition matrix $\mathbf{p}$:

$$\mathbf{p} \cdot P_{\text{inv}} = P_{\text{inv}}$$

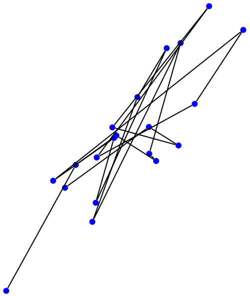
Continuous case

- $\mathbf{X}$ is now uncountable (e.g. $\mathbf{X} = \mathbb{R}^d$).
- The transition matrix $\mathbf{p}$ is substituted by the conditional probability t.
- A distribution P_{inv} with density p_{inv} is invariant if

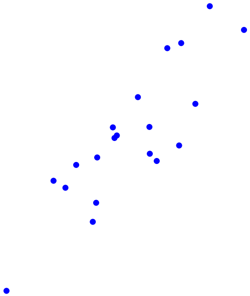
$$\int_{\mathbf{X}} t(y|x) p_{\text{inv}}(x) dx = p_{\text{inv}}(y)$$

This is simply the continuous analogue of the equation $\sum_i \mathbf{p}_{ij} (P_{\text{inv}})_i = (P_{\text{inv}})_j$.

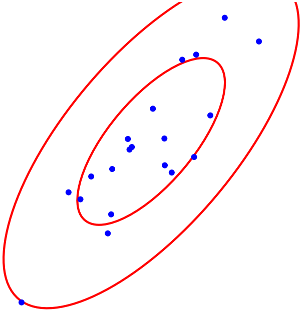
Markov Chain Sampling



We run the Markov chain n for steps. Each step moves from the current location x_i to a new x_{i+1}.



We "forget" the order and regard the locations $x_{1:n}$ as a random set of points.



If p (red contours) is both the invariant and initial distribution, each X_i is distributed as $X_i \sim p$.

Problems we need to solve

1. We have to construct a MC with invariant distribution p.
2. We cannot actually start sampling with $X_1 \sim p$; if we knew how to sample from p, all of this would be pointless.
3. Each point X_i is *marginally* distributed as $X_i \sim p$, but the points are *not* i.i.d.

CONSTRUCTING THE MARKOV CHAIN

Given is a continuous target distribution with density p.

Metropolis-Hastings (MH) kernel

1. We start by defining a conditional probability $q(y|x)$ on $\mathbf{X}$.
   q has nothing to do with p. We could e.g. choose $q(y|x) = g(y|x, \sigma^2)$, as in the previous example.
2. We define a **rejection kernel** A as

$$A(x_{i+1}|x_i) := \min\left\{1, \frac{q(x_i|x_{i+1})p(x_{i+1})}{q(x_{i+1}|x_i)p(x_i)}\right\}$$

   The normalization of p cancels in the quotient, so knowing $\tilde{p}$ is again enough.
3. We define the transition probability of the chain as

$$t(x_{i+1}|x_i) := q(x_{i+1}|x_i)A(x_{i+1}|x_i) + \delta_{x_i}(x_{i+1})c(x_i) \quad \text{where} \quad c(x_i) := \overbrace{\int q(y|x_i)(1-A(y|x_i))dy}^{\substack{\text{total probability that} \\ \text{a proposal is sampled} \\ \text{and then rejected}}}$$

Sampling from the MH chain

At each step $i+1$, generate a proposal $X^* \sim q(\,.\,|x_i)$ and $U_i \sim \text{Uniform}[0, 1]$.

- If $U_i \leq A(x^*|x_i)$, accept proposal: Set $x_{i+1} := x^*$.
- If $U_i > A(x^*|x_i)$, reject proposal: Set $x_{i+1} := x_i$.

STOCHASTIC HILL-CLIMBING

The Metropolis-Hastings rejection kernel was defined as:

$$A(x_{i+1}|x_i) = \min\left\{1, \frac{q(x_i|x_{i+1})p(x_{i+1})}{q(x_{i+1}|x_i)p(x_i)}\right\} .$$

Hence, we certainly accept if the second term is larger than 1, i.e. if

$$q(x_i|x_{i+1})p(x_{i+1}) > q(x_{i+1}|x_i)p(x_i) .$$

That means:

- We always accept the proposal value x_{i+1} if it *increases* the probability under p.
- If it *decreases* the probability, we still accept with a probability which depends on the difference to the current probability.

Hill-climbing interpretation

- The MH sampler somewhat resembles a gradient ascent algorithm on p, which *tends* to move in the direction of increasing probability p.
- However:
  - The actual steps are chosen at random.
  - The sampler can move "downhill" with a certain probability.
  - When it reaches a local maximum, it does not get stuck there.

Problem 1: Initial distribution

Recall: Fundamental theorem on Markov chains

Suppose we sample $X_1 \sim P_{\text{init}}$ and $X_{i+1} \sim t(\,.\,|x_i)$. This defines a distribution P_i of X_i, which can change from step to step. If the MC is nice (recall: irreducible and aperiodic), then

$$P_i \to P_{\text{inv}} \qquad \text{for} \qquad i \to \infty\ .$$

Note: Making precise what aperiodic means in a continuous state space is a bit more technical than in the finite case, but the theorem still holds. We will not worry about the details here.

Implication

- If we can show that $P_{\text{inv}} \equiv p$, we do not have to know how to sample from p.
- Instead, we can start with *any* P_{init}, and will get arbitrarily close to p for sufficiently large i.

Burn-In and Mixing Time

The number m of steps required until $P_m \approx P_{\text{inv}} \equiv p$ is called the **mixing time** of the Markov chain. (In probability theory, there is a range of definitions for what exactly $P_m \approx P_{\text{inv}}$ means.)

In MC samplers, the first m samples are also called the **burn-in** phase. The first m samples of each run of the sampler are discarded:

$$\underbrace{X_1, \dots, X_{m-1}}_{\substack{\text{Burn-in;} \\ \text{discard.}}}, \underbrace{X_m, X_{m+1}, \dots}_{\substack{\text{Samples from} \\ \text{(approximately) } p\text{;} \\ \text{keep.}}}$$

Convergence diagnostics

In practice, we do not know how large m is. There are a number of methods for assessing whether the sampler has mixed. Such heuristics are often referred to as **convergence diagnostics**.

PROBLEM 2: SEQUENTIAL DEPENDENCE

Even after burn-in, the samples from a MC are not i.i.d.

Strategy

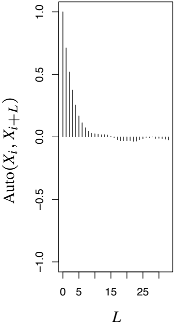
- Estimate empirically how many steps L are needed for x_i and x_{i+L} to be approximately independent. The number L is called the **lag**.
- After burn-in, keep only every Lth sample; discard samples in between.

Estimating the lag

The most commen method uses the **autocorrelation function**:

$$\text{Auto}(X_i, X_j) := \frac{\mathbb{E}[(X_i - \mu_i)(X_j - \mu_j)]}{\sigma_i \sigma_j} ,$$

where μ_i is the mean and σ_i the standard deviation of X_i. We compute $\text{Auto}(X_i, X_{i+L})$ empirically from the sample for different values of L, and find the smallest L for which the autocorrelation is close to zero.

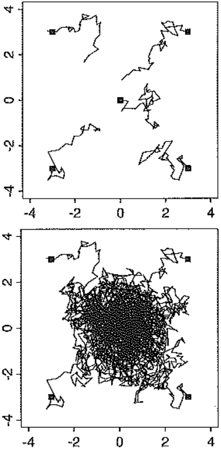


CONVERGENCE DIAGNOSTICS

There are about half a dozen popular convergence crieteria; the one below is an example.

Gelman-Rubin criterion

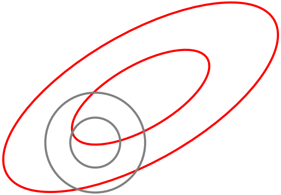
- Start several chains at random. For each chain k, sample X_i^k has a marginal distribution P_i^k.
- The distributions of P_i^k will differ between chains in early stages.
- Once the chains have converged, all $P_i = P_{\text{inv}}$ are identical.
- Criterion: Use a hypothesis test to compare P_i^k for different k (e.g. compare P_i^2 against null hypothesis P_i^1). Once the test does not reject anymore, assume that the chains are past burn-in.



Reference: A. Gelman and D. B. Rubin: "Inference from Iterative Simulation Using Multiple Sequences", *Statistical Science*, Vol. 7 (1992) 457-511.

SELECTING A PROPOSAL DISTRIBUTION

Everyone's favorite example: Two Gaussians



red = target distribution p
gray = proposal distribution q

- $\text{Var}[q]$ too large:
  Will overstep p; many rejections.
- $\text{Var}[q]$ too small:
  Many steps needed to achieve good coverage of domain.

If p is unimodal and can be roughly approximated by a Gaussian, $\text{Var}[q]$ should be chosen as smallest covariance component of p.

More generally

For complicated posteriors (recall: small regions of concentration, large low-probability regions in between) choosing q is much more difficult. To choose q with good performance, we already need to know something about the posterior.

There are many strategies, e.g. mixture proposals (with one component for large steps and one for small steps).

Summary: MH Sampler

- MCMC samplers construct a MC with invariant distribution p.
- The MH kernel is one generic way to construct such a chain from p and a proposal distribution q.
- Formally, q does not depend on p (but arbitrary choice of q usually means bad performance).
- We have to discard an initial number m of samples as burn-in to obtain samples (approximately) distributed according to p.
- After burn-in, we keep only every Lth sample (where L = lag) to make sure the x_i are (approximately) independent.

$$\underbrace{X_1,\ldots,X_{m-1}}_{\substack{\text{Burn-in;}\\ \text{discard.}}},\overset{\substack{\text{Keep.}\\ \downarrow}}{X_m},\underbrace{X_{m+1},\ldots,X_{m+L-1}}_{\substack{\text{Samples correlated}\\ \text{with } X_j\text{; discard.}}},\overset{\substack{\text{Keep.}\\ \downarrow}}{X_{m+L}},\underbrace{X_{m+L+1},\ldots X_{m+2L-1}}_{\substack{\text{Samples correlated}\\ \text{with } X_{j+L}\text{; discard.}}},\overset{\substack{\text{Keep.}\\ \downarrow}}{X_{m+2L}},\ldots$$

THE GIBBS SAMPLER

GIBBS SAMPLING

By far the most widely used MCMC algorithm is the Gibbs sampler.

Full conditionals

Suppose $\mathcal{L}(X)$ is a distribution on $\mathbb{R}^{\mathrm{D}}$, so $X = (X_1, \ldots, X_{\mathrm{D}})$. The conditional probability of the entry X_d given all other entries,

$$\mathcal{L}(X_d | X_1, \ldots, X_{d-1}, X_{d+1}, \ldots, X_{\mathrm{D}})$$

is called the **full conditional** distribution of X_d.

On $\mathbb{R}^{\mathrm{D}}$, that means we are interested in a density

$$p(x_d | x_1, \ldots, x_{d-1}, x_{d+1}, \ldots, x_{\mathrm{D}})$$

Gibbs sampling

The Gibbs sampler is the special case of the Metropolis-Hastings algorithm defined by

$$\text{propsoal distribution for } X_d \quad = \quad \text{full conditional of } X_d \; .$$

- Gibbs sampling is only applicable if we can compute the full conditionals for each dimension d.
- If so, it provides us with a *generic* way to derive a proposal distribution.

The Gibbs Sampler

Proposal distribution

Suppose p is a distribution on $\mathbb{R}^{\mathrm{D}}$, so each sample is of the form $X_i = (X_{i,1}, \dots, X_{i,\mathrm{D}})$. We generate a proposal X_{i+1} coordinate-by-coordinate as follows:

$$\begin{aligned}
X_{i+1,1} &\sim p(\,.\,|x_{i,2}, \dots, x_{i,\mathrm{D}}) \\
&\vdots \\
X_{i+1,d} &\sim p(\,.\,|x_{i+1,1}, \dots, x_{i+1,d-1}, x_{i,d+1}, \dots, x_{i,\mathrm{D}}) \\
&\vdots \\
X_{i+1,\mathrm{D}} &\sim p(\,.\,|x_{i+1,1}, \dots, x_{i+1,\mathrm{D}-1})
\end{aligned}$$

Note: Each new $X_{i+1,d}$ is *immediately* used in the update of the next dimension $d+1$.

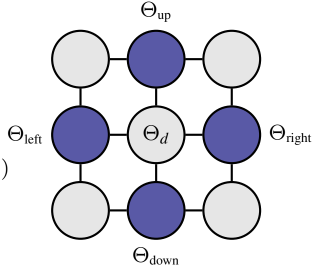
A Metropolis-Hastings algorithms with proposals generated as above is called a **Gibbs sampler**.

No rejections

One can show that the Metropolis-Hastings acceptance probability for each $x_{i+1,d+1}$ is 1, so *proposals in Gibbs sampling are always accepted*.

Example: MRF

In a MRF with D nodes, each dimension d corresponds to one vertex.



Full conditionals

In a grid with 4-neighborhoods for instance, the Markov property implies that

$$p(\theta_d|\theta_1,\ldots,\theta_{d-1},\theta_{d+1},\ldots,\theta_{\text{D}}) = p(\theta_d|\theta_{\text{left}},\theta_{\text{right}},\theta_{\text{up}},\theta_{\text{down}})$$

Specifically: Potts model with binary weights

Recall that, for sampling, knowing only $\tilde{p}$ (unnormalized) is sufficient:

$$\tilde{p}(\theta_d|\theta_1,\ldots,\theta_{d-1},\theta_{d+1},\ldots,\theta_{\text{D}}) =$$
$$\exp\Big(\beta(\mathbb{I}\{\theta_d = \theta_{\text{left}}\} + \mathbb{I}\{\theta_d = \theta_{\text{right}}\} + \mathbb{I}\{\theta_d = \theta_{\text{up}}\} + \mathbb{I}\{\theta_d = \theta_{\text{down}}\}\Big)$$

This is clearly very efficiently computable.

EXAMPLE: MRF

Sampling the Potts model

Each step of the sampler generates a sample

$$\theta_i = (\theta_{i,1}, \dots, \theta_{i,\mathrm{D}}) \ ,$$

where *D* is the number of vertices in the grid.

Gibbs sampler

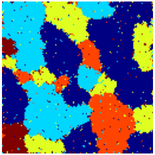
Each step of the Gibbs sampler generates *n* updates according to

$$\begin{aligned}\theta_{i+1,d} &\sim p(\,.\,|\theta_{i+1,1}, \dots, \theta_{i+1,d-1}, \theta_{i,d+1}, \dots, \theta_{i,\mathrm{D}}) \\ &\propto \exp\Big(\beta(\mathbb{I}\{\theta_{i+1,d} = \theta_{\text{left}}\} + \mathbb{I}\{\theta_{i+1,d} = \theta_{\text{right}}\} + \mathbb{I}\{\theta_{i+1,d} = \theta_{\text{up}}\} + \mathbb{I}\{\theta_{i+1,d} = \theta_{\text{down}}\})\Big)\end{aligned}$$

BURN-IN MATTERS

This example is due to Erik Sudderth (UC Irvine).

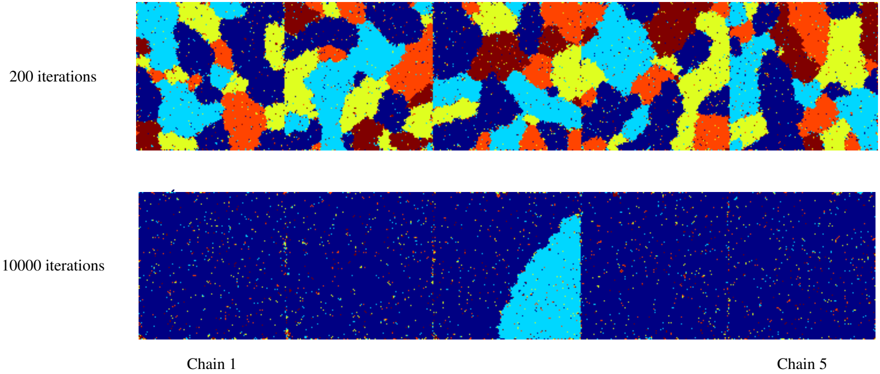
MRFs as "segmentation" priors



- MRFs where introduced as tools for image smoothing and segmentation by D. and S. Geman in 1984.
- They sampled from a Potts model with a Gibbs sampler, discarding 200 iterations as burn-in.
- Such a sample (after 200 steps) is shown above, for a Potts model in which each variable can take one out of 5 possible values.
- These patterns led computer vision researchers to conclude that MRFs are "natural" priors for image segmentation, since samples from the MRF resemble a segmented image.

EXAMPLE: BURN-IN MATTERS

E. Sudderth ran a Gibbs sampler on the same model and sampled after 200 iterations (as the Geman brothers did), and again after 10000 iterations:



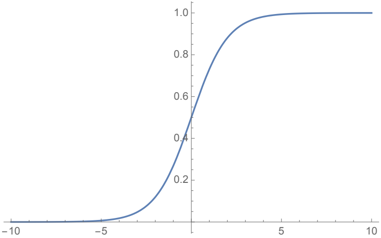
- The "segmentation" patterns are not sampled from the MRF distribution $p \equiv P_{\text{inv}}$, but rather from $P_{200} \neq P_{\text{inv}}$.
- The patterns occur not because MRFs are "natural" priors for segmentations, but because *the sampler's Markov chain has not mixed.*
- MRFs are smoothness priors, not segmentation priors.

TOOLS: LOGISTIC REGRESSION

Sigmoids

Sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

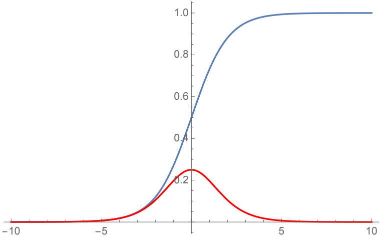


Note

$$1-\sigma(x) = \frac{1+e^{-x} \;-\; 1}{1+e^{-x}} = \frac{1}{e^{x}+1} = \sigma(-x)$$

Derivative

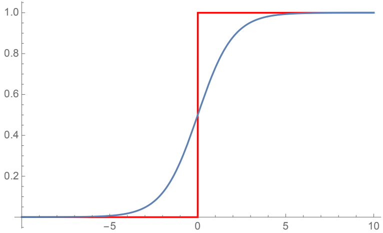
$$\frac{d\sigma}{dx}(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)\big(1-\sigma(x)\big)$$



Sigmoid (blue) and its derivative (red)

Approximating decision boundaries

- In linear classification: Decision boundary is a discontinuity
- Boundary is represented either by indicator function $\mathbb{I}\{\bullet > c\}$ or sign function $\operatorname{sign}(\bullet - c)$
- These representations are equivalent: Note $\operatorname{sign}(\bullet - c) = 2 \cdot \mathbb{I}\{\bullet > c\} - 1$



The most important use of the sigmoid function in machine learning is *as a smooth approximation to the indicator function*.

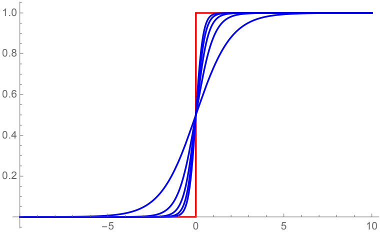
Given a sigmoid σ and a data point x, we decide which side of the approximated boundary we are own by thresholding

$$\sigma(x) \geq \frac{1}{2}$$

Scaling

We can add a scale parameter by definining

$$\sigma_\theta(x) := \sigma(\theta x) = \frac{1}{1 - e^{-\theta x}} \qquad \text{for } \theta \in \mathbb{R}$$



Influence of θ

- As θ increases, σ_θ approximates $\mathbb{I}$ more closely.
- For $\theta \to \infty$, the sigmoid converges to $\mathbb{I}$ pointwise, that is: For every $x \neq 0$, we have

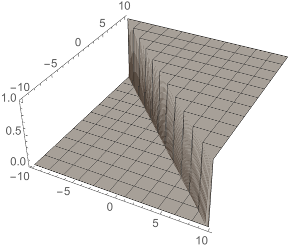
$$\sigma_\theta(x) \to \mathbb{I}\{x > 0\} \qquad \text{as } \theta \to +\infty \ .$$

- Note $\sigma_\theta(0) = \frac{1}{2}$ always, regardless of θ.

Approximating a Linear Classifier

So far, we have considered $\mathbb{R}$, but linear classifiers usually live in $\mathbb{R}^d$.

The decision boundary of a linear classifier in $\mathbb{R}^2$ is a discontinuous ridge:

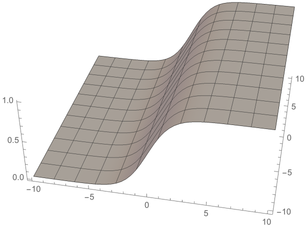


- This is a linear classifier of the form

$$\mathbb{I}\{\langle \mathbf{v}, \mathbf{x} \rangle - c\}.$$

- Here: $\mathbf{v} = (1, 1)$ and $c = 0$.

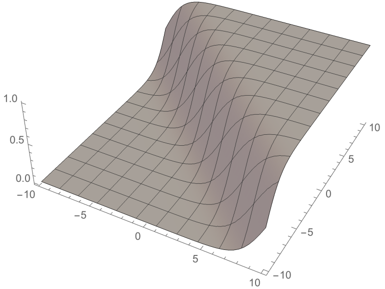
We can "stretch" σ into a ridge function on $\mathbb{R}^2$:

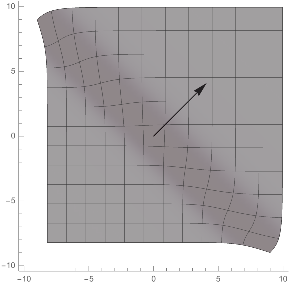


- This is the function $\mathbf{x} = (x_1, x_2) \mapsto \sigma(x_1)$.
- The ridge runs parallel to the x_2-axes.
- If we use $\sigma(x_2)$ instead, we rotate by 90 degrees (still axis-parallel).

Steering a Sigmoid

Just as for a linear classifier, we use a normal vector $\mathbf{v} \in \mathbb{R}^d$.





- The function $\sigma(\langle \mathbf{v}, \mathbf{x} \rangle - c)$ is a sigmoid ridge, where the ridge is orthogonal to the normal vector $\mathbf{v}$, and c is an offset that shifts the ridge "out of the origin".
- The plot on the right shows the normal vector (here: $\mathbf{v} = (1, 1)$) in black.
- The parameters $\mathbf{v}$ and c have the same meaning for $\mathbb{I}$ and σ, that is, $\sigma(\langle \mathbf{v}, \mathbf{x} \rangle - c)$ approximates $\mathbb{I}\{\langle \mathbf{v}, \mathbf{x} \rangle \geq c\}$.

LOGISTIC REGRESSION

Logistic regression is a classification method that approximates decision boundaries by sigmoids.

Setup

- Two-class classification problem
- Observations $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$, class labels $y_i \in \{0, 1\}$.

The logistic regression model

We model the conditional distribution of the class label given the data as

$$P(y|\mathbf{x}) := \text{Bernoulli}\big(\sigma(\langle \mathbf{v}, \mathbf{x} \rangle - c)\big) \; .$$

- Recall $\sigma(\langle \mathbf{v}, \mathbf{x} \rangle - c)$ takes values in $[0, 1]$ for all θ, and value $\frac{1}{2}$ on the class boundary.
- The logistic regression model interprets this value as the probability of being in class y.

Learning Logistic Regression

Since the model is defined by a parametric distribution, we can apply maximum likelihood.

Notation

Recall from Statistical Machine Learning: We collect the parameters in a vector **w** by writing

$$\mathbf{w} := \begin{pmatrix} \mathbf{v} \\ -c \end{pmatrix} \qquad \text{and} \qquad \tilde{\mathbf{x}} := \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \qquad \text{so that } \langle \mathbf{w}, \tilde{\mathbf{x}} \rangle = \langle \mathbf{v}, \mathbf{x} \rangle - c \; .$$

Likelihood function of the logistic regression model

$$\prod_{i=1}^{n} \sigma(\langle \mathbf{w}, \tilde{\mathbf{x}}_i \rangle)^{y_i} \big(1 - \sigma(\langle \mathbf{w}, \tilde{\mathbf{x}}_i \rangle)\big)^{1-y_i}$$

Negative log-likelihood

$$L(\mathbf{w}) \quad := \quad -\sum_{i=1}^{n} \Big(y_i \log \sigma(\langle \mathbf{w}, \tilde{\mathbf{x}}_i \rangle) + (1 - y_i) \log\big(1 - \sigma(\langle \mathbf{w}, \tilde{\mathbf{x}}_i \rangle)\big) \Big)$$

Maximum Likelihood

$$\nabla L(\mathbf{w}) \quad = \quad \sum_{i=1}^{n} \big(\sigma(\mathbf{w}^t \tilde{\mathbf{x}}_i) - y_i\big)\tilde{\mathbf{x}}_i$$

Note

- Each training data point $\mathbf{x}_i$ contributes to the sum proportionally to the approximation error $\sigma(\mathbf{w}^t \tilde{\mathbf{x}}_i) - y_i$ incurred at $\mathbf{x}_i$ by approximating the linear classifier by a sigmoid.

Maximum likelihood

- The ML estimator $\hat{\mathbf{w}}$ for $\mathbf{w}$ is the solution of
$$\nabla L(\mathbf{w}) = 0 \; .$$
- For logistic regression, this equation has no solution in closed form.
- To find $\hat{\mathbf{w}}$, we use numerical optimization.
- The function L is convex (= $\cup$-shaped).

Recall from Statistical Machine Learning

- If f is differentiable, we can apply gradient descent:

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \nabla f(\mathbf{x}^{(k)})$$

  where $\mathbf{x}^{(k)}$ is the candidate solution in step k of the algorithm.

- If the Hessian matrix H_f of partial second derivatives exists and is invertible, we can apply Newton's method, which converges faster:

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - H_f^{-1}(\mathbf{x}^{(k)}) \cdot \nabla f(\mathbf{x}^{(k)})$$

- Recall that the Hessian matrix of a (twice continuously differentiable) function $f : \mathbb{R}^d \to \mathbb{R}$ is

$$H_f(\mathbf{x}) := \Big(\frac{\partial^2 f}{\partial x_i \partial x_j} \Big)_{i,j \leq n}$$

  Since f is twice differentiable, each $\partial^2 f / \partial x_i \partial x_j$ exists; since it is twice *continuously* differentiable, $\partial^2 f / \partial x_i \partial x_j = \partial^2 f / \partial x_j \partial x_i$, so H_f is symmetric.

- The inverse of $H_f(\mathbf{x})$ exists if and only if the matrix is positive definite (semidefinite does not suffice), which in turn is true if and only if f is strictly convex.

NEWTON'S METHOD FOR LOGISTIC REGRESSION

Applying Newton

$$\mathbf{w}^{(k+1)} := \mathbf{w}^{(k)} - H_L^{-1}(\mathbf{w}^{(k)}) \cdot \nabla L(\mathbf{w}^{(k)})$$

Matrix notation

$$\tilde{\mathbf{X}} := \left(\begin{array}{cccccc} 1 & (\tilde{\mathbf{x}}_1)_1 & \dots & (\tilde{\mathbf{x}}_1)_j & \dots & (\tilde{\mathbf{x}}_1)_d \\ \vdots & & & \vdots & & \vdots \\ 1 & \multicolumn{5}{c}{\boxed{(\tilde{\mathbf{x}}_i)_1 \quad \dots \quad (\tilde{\mathbf{x}}_i)_j \quad \dots \quad (\tilde{\mathbf{x}}_i)_d}} \\ \vdots & & & \vdots & & \vdots \\ 1 & (\tilde{\mathbf{x}}_n)_1 & \dots & (\tilde{\mathbf{x}}_n)_j & \dots & (\tilde{\mathbf{x}}_n)_d \end{array}\right) \longleftarrow \tilde{\mathbf{x}}_i \qquad D_\sigma = \begin{pmatrix} \sigma(\mathbf{w}^t\tilde{\mathbf{x}}_1)(1-\sigma(\mathbf{w}^t\tilde{\mathbf{x}}_1)) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma(\mathbf{w}^t\tilde{\mathbf{x}}_n)(1-\sigma(\mathbf{w}^t\tilde{\mathbf{x}}_n)) \end{pmatrix}$$

$\tilde{\mathbf{X}}$ is the data matrix (or design matrix) you know from linear regression. $\tilde{\mathbf{X}}$ has size $n \times (d+1)$ and D_σ is $n \times n$.

Newton step

$$\mathbf{w}^{(k+1)} = \left(\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}\right)^{-1} \tilde{\mathbf{X}}^t D_\sigma \left(\tilde{\mathbf{X}}\mathbf{w}^{(k)} - D_\sigma\left(\begin{pmatrix} \sigma(\langle \mathbf{w}^{(k)}, \tilde{\mathbf{x}}_1 \rangle) \\ \vdots \\ \sigma(\langle \mathbf{w}^{(k)}, \tilde{\mathbf{x}}_n \rangle) \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}\right)\right)$$

Newton's Method for Logistic Regression

$$\mathbf{w}^{(k+1)} = \big(\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}\big)^{-1} \tilde{\mathbf{X}}^t D_\sigma \underbrace{\Bigg(\tilde{\mathbf{X}}\mathbf{w}^{(k)} - D_\sigma\Big(\begin{pmatrix}\sigma(\langle \mathbf{w}^{(k)}, \tilde{\mathbf{x}}_1\rangle) \\ \vdots \\ \sigma(\langle \mathbf{w}^{(k)}, \tilde{\mathbf{x}}_n\rangle)\end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}\Big)\Bigg)}_{=:\, \mathbf{u}^{(k)}} = \big(\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}\big)^{-1} \tilde{\mathbf{X}}^t D_\sigma \mathbf{u}^{(k)}$$

Compare this to the least squares solution of a linear regression problem:

$$\hat{\boldsymbol{\beta}} = (\tilde{\mathbf{X}}^t \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^t y \qquad\qquad \mathbf{w}^{(k+1)} = \big(\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}\big)^{-1} \tilde{\mathbf{X}}^t D_\sigma \mathbf{u}^{(k)}$$

Differences:

- The vector y of regression responses is substituted by the vector $\mathbf{u}^{(k)}$ above.
- The matrix $\tilde{\mathbf{X}}^t \tilde{\mathbf{X}}$ is substituted by the matrix $\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}$.
- Note that matrices of product form $\tilde{\mathbf{X}}^t \tilde{\mathbf{X}}$ are positive semidefinite; since D_σ is diagonal with non-negative entries, so is $\tilde{\mathbf{X}}^t D_\sigma \tilde{\mathbf{X}}$.

Iteratively Reweighted Least Squares

- At each step, the algorithm solves a least-squares problem "reweighted" by the matrix D_σ.
- Since this happens at each step of an iterative algorithm, Newton's method applied to the logistic regression log-likelihood is also known as *Iteratively Reweighted Least Squares*.

OTHER OPTIMIZATION METHODS

Newton: Cost

- The size of the Hessian is $(d+1) \times (d+1)$.
- In high-dimensional problems, inverting H_L can become problematic.

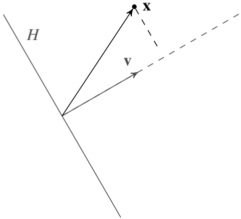
Other methods

Maximum likelihood only requires that we minimize the negative log-likelihood; we can choose any numerical method, not just Newton. Alternatives include:

- Pseudo-Newton methods (only invert H_L once, for $\mathbf{w}^{(1)}$, but do not guarantee quadratic convergence).
- Gradient methods.
- Approximate gradient methods, like stochastic gradient.

OVERFITTING

Recall from Statistical Machine Learning

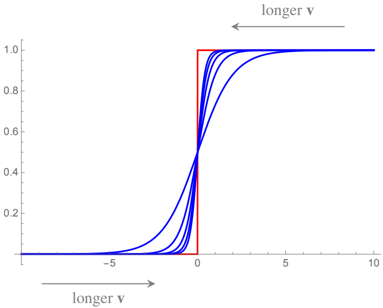


- If we increase the length of $\mathbf{v}$ without changing its direction, the sign of $\langle \mathbf{v}, \mathbf{x} \rangle$ does not change, but the value changes.
- That means: If $\mathbf{v}$ is the normal vector of a classifier, and we scale $\mathbf{v}$ by some $\theta > 0$, the decision boundary does not move, but $\langle \theta\mathbf{v}, \mathbf{x} \rangle = \theta \langle \mathbf{v}, \mathbf{x} \rangle$.

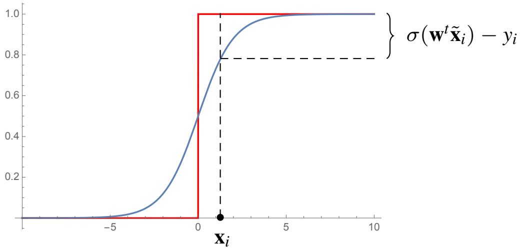
Effect inside a sigmoid

$$\sigma(\langle \theta\mathbf{v}, \mathbf{x} \rangle) = \sigma(\theta \langle \mathbf{v}, \mathbf{x} \rangle) = \sigma_\theta(\langle \mathbf{v}, \mathbf{x} \rangle)$$

As the length of $\mathbf{v}$ increases, $\sigma(\langle \mathbf{v}, \mathbf{x} \rangle)$ becomes more similar to $\mathbb{I}\{\langle \mathbf{v}, \mathbf{x} \rangle > 0\}$.



EFFECT ON ML FOR LOGISTIC REGRESSION



- Recall each training data point $\mathbf{x}_i$ contributes an error term $\sigma(\mathbf{w}^t\tilde{\mathbf{x}}_i) - y_i$ to the log-likelihood.
- By increasing the lenghts of $\mathbf{w}$, we can make $\sigma(\mathbf{w}^t\tilde{\mathbf{x}}_i) - y_i$ arbitrarily small without moving the decision boundary.

OVERFITTING

Consequence for linearly separable data

- Once the decision boundary is correctly located between the two classes, the maximization algorithm can increase the log-likelihood arbitrarily by increasing the length of **w**.
- That does not move the decision boundary, but he logistic function looks more and more like the indicator $\mathbb{I}$.
- That may fit the training data more tightly, but can lead to bad generalization (e.g. for similar reasons as for the perceptron, where the decision boundary may end up very close to a training data point).

That is a form of overfitting.

Data that is not linearly separable

- If the data is not separable, sufficiently many points on the "wrong" side of the decision boundary prevent overfitting (since making **w** larger *increases* error contributions of these points).
- For large data sets, overfitting can still occur if the fraction of such points is small.

Solutions

- Overfitting can be addressed by including an additive penalty of the form $L(\mathbf{w}) + \lambda \|\mathbf{w}\|$.

Logistic Regression for Multiple Classes

Bernoulli and multinomial distributions

- The mulitnomial distribution of N draws from K categories with parameter vector $(\theta_1, \ldots, \theta_K)$ (where $\sum_{k \leq K} \theta_k = 1$) has probabililty mass function

$$P(m_1, \ldots, m_K | \theta_1, \ldots, \theta_K) = \frac{N!}{m_1! \cdots m_K!} \prod_{k=1}^{K} \theta_k^{m_k} \qquad \text{where } m_k = \text{\# draws in category } k$$

- Note that $\text{Bernoulli}(p) = \text{Multinomial}(p, 1-p; N = 1)$.

Logistic regression

- Recall two-class logistic regression is defined by $P(Y|\mathbf{x}) = \text{Bernoulli}(\sigma(\mathbf{w}^t\mathbf{x}))$.
- Idea: To generalize logistic regression to K classes, choose a separate weight vector $\mathbf{w}_k$ for each class k, and define $P(Y|\mathbf{x})$ by

$$\text{Multinomial}\big(\tilde{\sigma}(\mathbf{w}_1^t\mathbf{x}), \ldots, \tilde{\sigma}(\mathbf{w}_K^t\mathbf{x})\big)$$

where $\tilde{\sigma}(\mathbf{w}_1^t\mathbf{x}) = \frac{\sigma(\mathbf{w}_1^t\mathbf{x})}{\sum_k \sigma(\mathbf{w}_k^t\mathbf{x})}$.

Logistic Regression for Multiple Classes

Logistic regression for K classes

The label y now takes values in $\{1, \dots, K\}$.

$$P(y|\mathbf{x}) = \prod_{k=1}^{K} \tilde{\sigma}(\mathbf{w}_k^t \tilde{\mathbf{x}})^{\mathbb{I}\{y=k\}}$$

The negative log-likelihood becomes

$$L(\mathbf{w}_1, \dots, \mathbf{w}_K) \quad = \quad -\sum_{i \leq n,\, k \leq K} \mathbb{I}\{y = k\} \log \tilde{\sigma}(\mathbf{w}_k^t \tilde{x}_i)$$

This can again be optimized numerically.

Comparison to two-class case

- Recall that $1 - \sigma(x) = \sigma(-x)$.
- That means

$$\text{Bernoulli}\big(\sigma(\langle \mathbf{v}, \mathbf{x} \rangle - c)\big) \equiv \text{Multinomial}\big(\sigma(\mathbf{w}^t \mathbf{x}), \sigma((-1)\mathbf{w}^t \mathbf{x})\big)$$

- That is: Two-class logistic regression as above is equivalent to multiclass logistic regression with $K = 2$ *provided* we choose $\mathbf{w}_2 = -\mathbf{w}_1$.

NEURAL NETWORKS

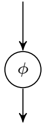
THE MOST IMPORTANT BIT

A neural network represents a function $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$.

BUILDING BLOCKS

Units

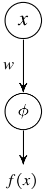
The basic building block is a **node** or **unit**:



- The unit has incoming and outgoing arrows. We think of each arrow as "transmitting" a signal.
- The signal is always a scalar.
- A unit represents a function ϕ.

We read the diagram as: A scalar value (say x) is transmitted to the unit, the function ϕ is applied, and the result $\phi(x)$ is transmitted from the unit along the outgoing arrow.

Weights

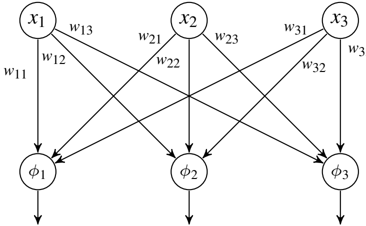


- If we want to "input" a scalar x, we represent it as a unit, too.
- We can think of this as the unit representing the constant function $g(x) = x$.
- Additionally, each arrow is usually inscribed with a (scalar) weight w.
- As the signal x passes along the edge, it is multiplied by the edge weight w.

The diagram above represents the function $f(x) := \phi(wx)$.

READING NEURAL NETWORKS

$$f : \mathbb{R}^3 \to \mathbb{R}^3 \quad \text{with input} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_2 \end{pmatrix}$$



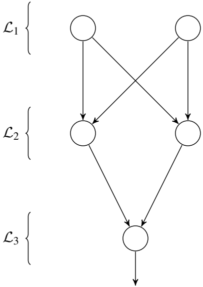
$f_1(\mathbf{x}) = \phi_1(\langle \mathbf{w}_1, \mathbf{x} \rangle)$ $f_2(\mathbf{x}) = \phi_2(\langle \mathbf{w}_2, \mathbf{x} \rangle)$ $f_3(\mathbf{x}) = \phi_3(\langle \mathbf{w}_3, \mathbf{x} \rangle)$

$$f(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \end{pmatrix} \quad \text{with} \quad f_i(\mathbf{x}) = \phi_i\Big(\sum_{j=1}^{3} w_{ji} x_j\Big)$$

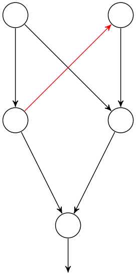
FEED-FORWARD NETWORKS

A **feed-forward network** is a neural network whose units can be arranged into groups $\mathcal{L}_1, \ldots, \mathcal{L}_K$ so that connections (arrows) only pass from units in group $\mathcal{L}_k$ to units in group $\mathcal{L}_{k+1}$. The groups are called **layers**. In a feed-forward network:

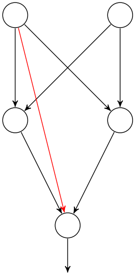
- There are no connections within a layer.
- There are no backwards connections.
- There are no connections that skip layers, e.g. from $\mathcal{L}_k$ to units in group $\mathcal{L}_{k+2}$.



feed-forward

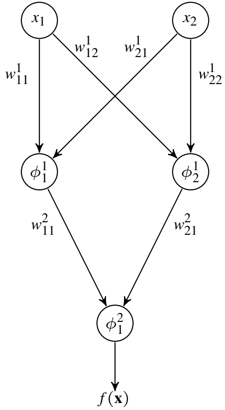


not feed-forward



not feed-forward

LAYERS



- This network computes the function

$$f(x_1,x_2) = \phi^2_1\Big(w^2_{11}\phi^1_1\big(w^1_{11}x_1+w^1_{21}x_2\big)+w^2_{12}\phi^1_2\big(w^1_{21}x_1+w^1_{22}x_2\big)\Big)$$

- Clearly, writing out f gets complicated fairly quickly as the network grows.

First shorthand: Scalar products

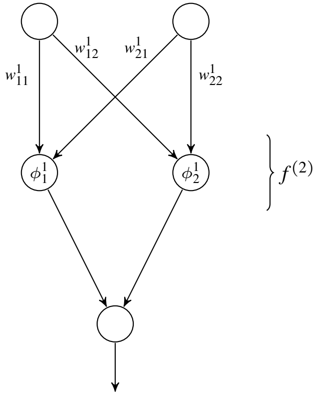
- Collect all weights coming into a unit into a vector, e.g.

$$\mathbf{w}^2_1 := (w^2_{11}, w^2_{21})$$

- Same for inputs: $\mathbf{x} = (x_1, x_2)$
- The function then becomes

$$f(\mathbf{x}) = \phi^2_1\left(\left\langle \mathbf{w}^2_1, \begin{pmatrix}\phi^1_1(\langle \mathbf{w}^1_1, \mathbf{x}\rangle)\\ \phi^1_2(\langle \mathbf{w}^1_2, \mathbf{x}\rangle)\end{pmatrix}\right\rangle\right)$$

LAYERS



- Each layer represents a function, which takes the output values of the previous layers as its arguments.
- Suppose the output values of the two nodes at the top are y_1, y_2.
- Then the second layer defines the (two-dimensional) function

$$f^{(2)}(\mathbf{y}) = \begin{pmatrix} \phi^1_1(\langle \mathbf{w}^1_1, \mathbf{y} \rangle) \\ \phi^1_2(\langle \mathbf{w}^1_2, \mathbf{y} \rangle) \end{pmatrix}$$

Composition of Functions

Basic composition

Suppose f and g are two function $\mathbb{R} \to \mathbb{R}$. Their **composition** $g \circ f$ is the function

$$g \circ f(x) := g(f(x)) \ .$$

For example:

$$f(x) = x + 1 \qquad g(y) = y^2 \qquad g \circ f(x) = (x+1)^2$$

We could combine the same functions the other way around:

$$f \circ g(x) = x^2 + 1$$

In multiple dimensions

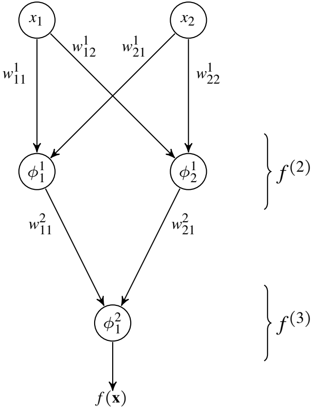
Suppose $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ and $g : \mathbb{R}^{d_2} \to \mathbb{R}^{d_3}$. Then

$$g \circ f(\mathbf{x}) = g(f(\mathbf{x})) \qquad \text{is a function } \mathbb{R}^{d_1} \to \mathbb{R}^{d_3} \ .$$

For example:

$$f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{v} \rangle - c \qquad g(y) = \operatorname{sgn}(y) \qquad g \circ f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{v} \rangle - c)$$

LAYERS AND COMPOSITION



- As above, we write

$$f^{(2)}(\bullet) = \begin{pmatrix} \phi_1^1(\langle \mathbf{w}_1^1, \bullet \rangle) \\ \phi_2^1(\langle \mathbf{w}_2^1, \bullet \rangle) \end{pmatrix}$$

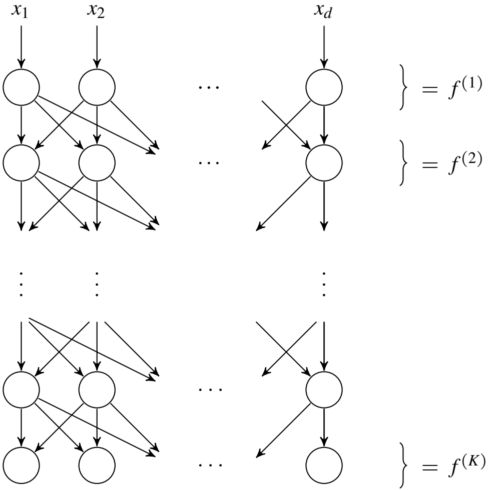
- The function for the third layer is similarly

$$f^{(3)}(\bullet) = \phi_1^2(\left\langle \mathbf{w}_1^2, \bullet \right\rangle)$$

- The entire network represents the function

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(\mathbf{x})) = f^{(3)} \circ f^{(2)}(\mathbf{x})$$

A feed-forward network represents a function as a composition of several functions, each given by one layer.



$$f(\mathbf{x}) \quad = \quad f^{(K)}(\cdots f^{(2)}(f^{(1)}(\mathbf{x}))) \quad = \quad f^{(K)} \circ \ldots \circ f^{(1)}(\mathbf{x})$$

LAYERS AND COMPOSITIONS

General feed-forward networks

A feed-forward network with K layers represents a function

$$f(\mathbf{x}) \quad = \quad f^{(K)} \circ \ldots \circ f^{(1)}(\mathbf{x})$$

Each layer represents a function $f^{(k)}$. These functions are of the form:

$$f^{(k)}(\,\bullet\,) = \begin{pmatrix} \phi_1^{(k)}(\langle \mathbf{w}_1^{(k)}, \bullet \rangle) \\ \vdots \\ \phi_d^{(k)}(\langle \mathbf{w}_d^{(k)}, \bullet \rangle) \end{pmatrix} \qquad \text{typically:} \qquad \phi^{(k)}(x) = \begin{cases} \sigma(x) & \text{(sigmoid)} \\ \mathbb{I}\{\pm x > \tau\} & \text{(threshold)} \\ c & \text{(constant)} \\ x & \text{(linear)} \\ \max\{0, x\} & \text{(rectified linear)} \end{cases}$$

Dimensions

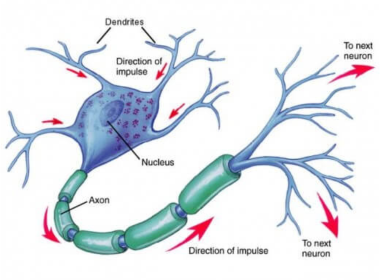
- Each function $f^{(k)}$ is of the form

$$f^{(k)} : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$$

- d_k is the number of nodes in the kth layer. It is also called the *width* of the layer.
- We mostly assume for simplicity: $d_1 = \ldots = d_K =: d$.

Origin of the Name

If you look up the term “neuron” online, you will find illustrations like this:



This one comes from a web site called easyscienceforkids.com, which means it is likely to be scientifically more accurate than typical references to “neuron” and “neural” in machine learning.

Roughly, a neuron is a brain cell that:

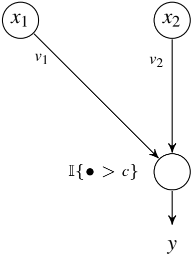
- Collects electrical signals (typically from other neurons)
- Processes them
- Generates an output signal

What happens inside a neuron is an intensely studied problem in neuroscience.

Source: easyscienceforkids.com

HISTORICAL PERSPECTIVE: MCCULLOCH-PITTS NEURON

A neuron is modeled as a “thresholding device” that combines input signals:



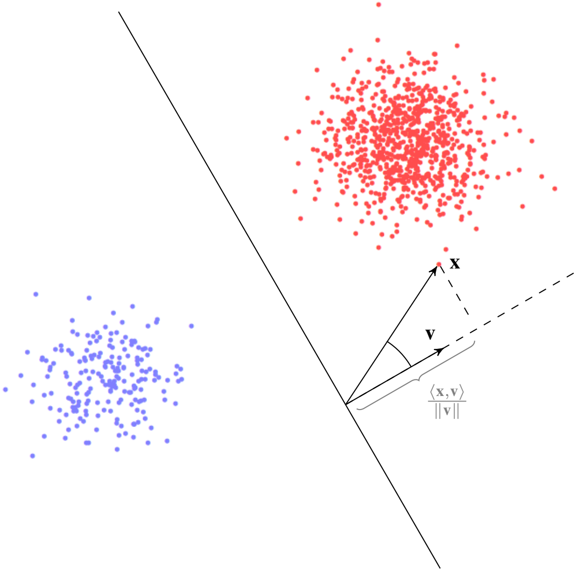
McCulloch-Pitts neuron model (1943)

- Collect the input signals x_1, x_2 into a vector $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$
- Choose fixed vector $\mathbf{v} \in \mathbb{R}^2$ and constant $c \in \mathbb{R}$.
- Compute:

$$y = \mathbb{I}\{\langle \mathbf{v}, \mathbf{x} \rangle > c\} \qquad \text{for some } c \in \mathbb{R} \ .$$

- In hindsight, this is a neural network with two layers, and function $\phi(\,\bullet\,) = \mathbb{I}\{\,\bullet\, > c\}$ at the bottom unit.

RECALL: LINEAR CLASSIFICATION



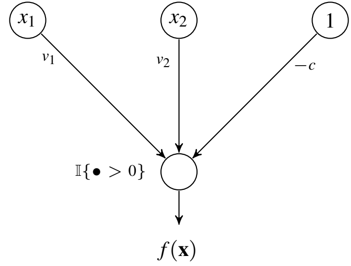
$\mathbf{x}$

$\mathbf{v}$

$\frac{\langle \mathbf{x}, \mathbf{v} \rangle}{\|\mathbf{v}\|}$

$$f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{v}, \mathbf{x} \rangle - c)$$

LINEAR CLASSIFIER IN $\mathbb{R}^2$ AS TWO-LAYER NN



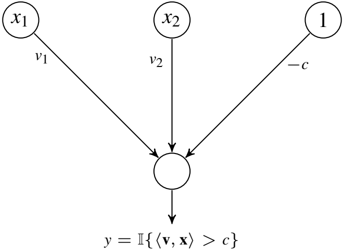
$$f(\mathbf{x}) \quad = \quad \mathbb{I}\{\, v_1x_1 + v_2x_2 + 1 \cdot (-c) \;>\; 0 \,\} \quad = \quad \mathbb{I}\{\langle \mathbf{v}, \mathbf{x}\rangle > c\}$$

Equivalent to linear classifier

The linear classifier on the previous slide and f differ only in whether they encode the “blue” class as -1 or as 0:

$$\operatorname{sgn}(\langle \mathbf{v}, \mathbf{x}\rangle - c) \;=\; 2f(\mathbf{x}) - 1$$

REMARKS

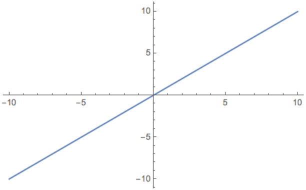

x_1
x_2
1
v_1
v_2
$-c$
$y = \mathbb{I}\{\langle \mathbf{v}, \mathbf{x} \rangle > c\}$

- This neural network represents a linear two-class classifier (on $\mathbb{R}^2$).
- We can more generally define a classifier on $\mathbb{R}^d$ by adding input units, one per dimension.
- It does not specify the training method.
- To train the classifier, we need a cost function and an optimization method.

Typical Component Functions

Linear units

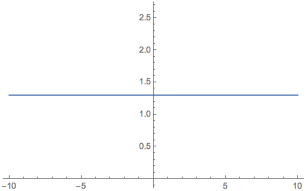
$$\phi(x) = x$$



This function simply “passes on” its incoming signal. These are used for example to represent inputs (data values).

Constant functions

$$\phi(x) = c$$

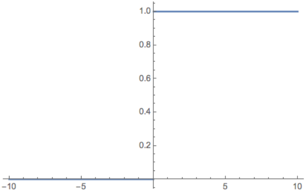


These can be used e.g. in combination with an indicator function to define a threshold, as in the linear classifier above.

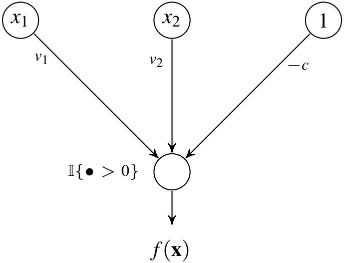
Typical Component Functions

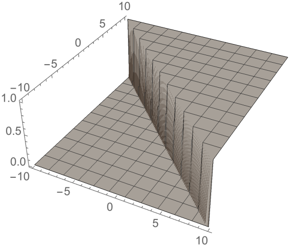
Indicator function

$$\phi(x) = \mathbb{I}\{x > 0\}$$



Example: Final unit is indicator

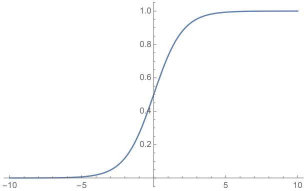




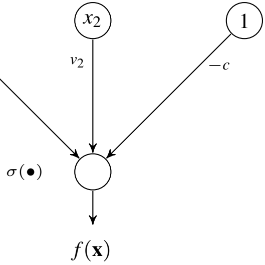
Typical Component Functions

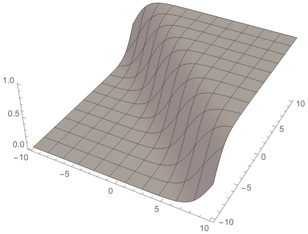
Sigmoids

$$\phi(x) = \frac{1}{1 + e^{-x}}$$



Example: Final unit is sigmoid

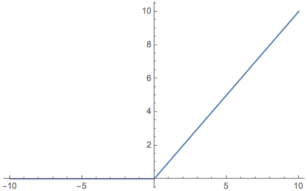




TYPICAL COMPONENT FUNCTIONS

Rectified linear units

$$\phi(x) = \max\{0, x\}$$



These are currently perhaps the most commonly used unit in the “inner” layers of a neural network (those layers that are not the input or output layer).

Hidden Layers and Nonlinear Functions

Hidden units

- Any nodes (or "units") in the network that are neither input nor output nodes are called **hidden**.
- Every network has an input layer and an output layer.
- If there any additional layers (which hence consist of hidden units), they are called **hidden layers**.

Linear and nonlinear networks

- If a network has no hidden units, then

$$f_i(\mathbf{x}) = \phi_i(\langle \mathbf{w}_i, \mathbf{x} \rangle)$$

  That means: f is a linear functions, except perhaps for the final application of ϕ.

- For example: In a classification problem, a two layer network can only represent linear decision boundaries.
- Networks with at least one hidden layer can represent nonlinear decision surfaces.

Two vs Three Layers

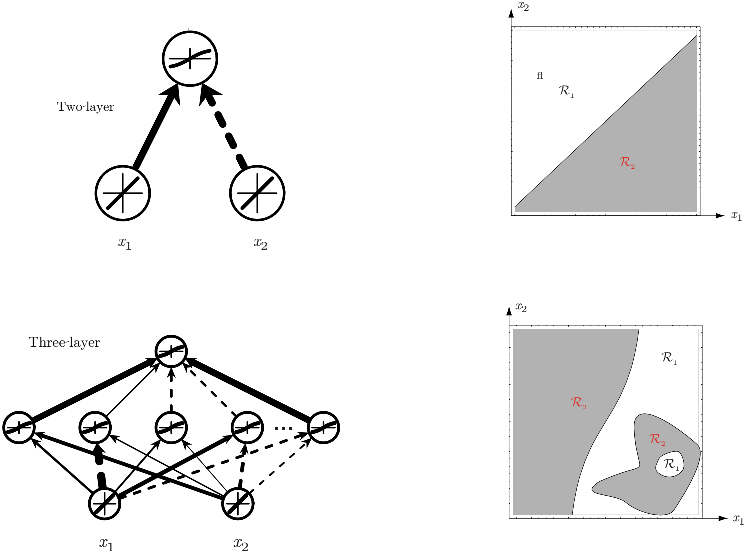
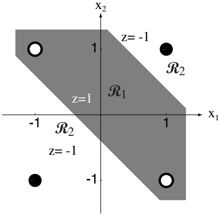
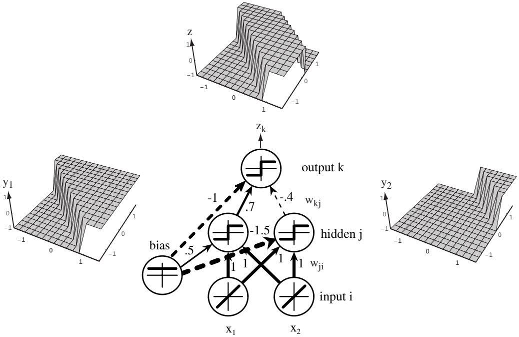


Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

THE XOR PROBLEM



Solution regions we would like to represent



Neural network representation

- Two ridges at different locations are substracted from each other.
- That generates a region bounded on both sides.
- A linear classifier cannot represent this decision region.
- Note this requires at least one hidden layer.

Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

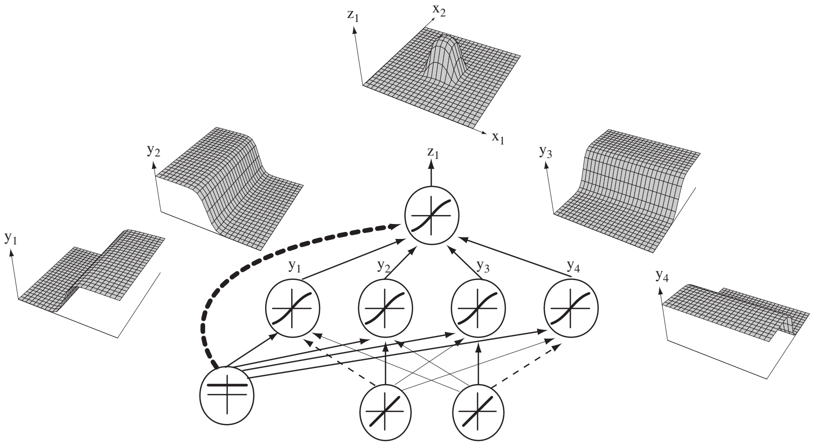


Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

NUMBER OF LAYERS

We have observed

- We have seen that two-layer classification networks always represent linear class boundaries.
- With three layers, the boundaries can be non-linear.

Obvious question

- What happens if we use more than three layers? Do four layers again increase expressive power?

Not examinable.

WIDTH VS DEPTH

A neural network represents a (typically) complicated function f by simple functions $\phi_i^{(k)}$.

What functions can be represented?

A well-known result in approximation theory says: Every continuous function $f : [0,1]^d \to \mathbb{R}$ can be represented in the form

$$f(\mathbf{x}) = \sum_{j=1}^{2d+1} \xi_j\Big(\sum_{i=1}^{d} \tau_{ij}(x_i)\Big)$$

where ξ_i and τ_{ij} are functions $\mathbb{R} \to \mathbb{R}$. A similar result shows one can approximate f to arbitrary precision using specifically sigmoids, as

$$f(\mathbf{x}) \approx \sum_{j=1}^{M} w_j^{(2)} \sigma\Big(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + c_i\Big)$$

for some finite M and constants c_i.

Note the representations above can both be written as neural networks with three layers (i.e. with one hidden layer).

Width vs Depth

Depth rather than width

- The representations above can achieve arbitrary precision with a single hidden layer (roughly: a three-layer neural network can represent any continuous function).
- In the first representation, ξ_j and τ_{ij} are "simpler" than f because they map $\mathbb{R} \to \mathbb{R}$.
- In the second representation, the functions are more specific (sigmoids), and we typically need more of them (M is large).
- That means: The price of precision are many hidden units, i.e. the network grows wide.
- The last years have shown: We can obtain very good results by limiting layer width, and instead increasing depth (= number of layers).
- There is no coherent theory yet to properly explain this behavior.

Limiting width

- Limiting layer width means we limit the degrees of freedom of each function $f^{(k)}$.
- That is a notion of parsimony.
- Again: There seem to be a lot of interesting questions to study here, but so far, we have no real answers.

Not examinable.

TRAINING NEURAL NETWORKS

Task

- We decide on a neural network "architecture": We fix the network diagram, including all functions ϕ at the units. Only the weights w on the edges can be changed during by training algorithm. Suppose the architecture we choose has d_1 input units and d_2 output units.
- We collect all weights into a vector $\mathbf{w}$. The entire network then represents a function $f_{\mathbf{w}}(\mathbf{x})$ that maps $\mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$.
- To "train" the network now means that, given training data, we have to determine a suitable parameter vector $\mathbf{w}$, i.e. we fit the network to data by fitting the weights.

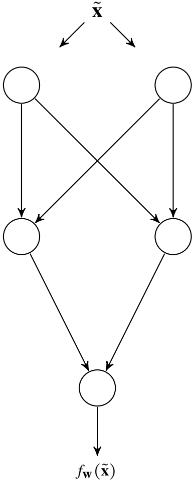
More specifically: Classification

Suppose the network is meant to represent a two-class classifier.

- That means the output dimension is $d_2 = 1$, so $f_{\mathbf{w}}$ is a function $\mathbb{R}^{d_1} \rightarrow \mathbb{R}$.
- We are given data $\mathbf{x}_1, \mathbf{x}_2, \ldots$ with labels $y_1, y_2, \ldots$.
- We split this data into training, validation and test data, according to the requirements of the problem we are trying to solve.
- We then fit the network to the training data.

Not examinable.

TRAINING NEURAL NETWORKS



- We run each training data point $\tilde{\mathbf{x}}_i$ through the network $f_{\mathbf{w}}$ and compare $f_{\mathbf{w}}(\tilde{\mathbf{x}}_i)$ to $\tilde{y}_i$ to measure the error.
- Recall how gradient descent works: We make "small" changes to **w**, and choose the one which decreases the error most. That is one step of the gradient scheme.
- For each such changed value $\mathbf{w}'$, we again run each training data point $\tilde{\mathbf{x}}_i$ through the network $f_{\mathbf{w}'}$, and measure the error by comparing $f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i)$ to $\tilde{y}_i$.

Not examinable.

TRAINING NEURAL NETWORKS

Error measure

- We have to specify how we compare the network's output $f_{\mathbf{w}}(\mathbf{x})$ to the correct answer y.
- To do so, we specify a function D with two arguments that serves as an error measure.
- The choice of D depends on the problem.

Typical error measures

- Classification problem:

$$D(\hat{y}, y) := y \log \hat{y} \qquad \text{(with convention } 0 \log 0 = 0\text{)}$$

- Regression problem:

$$D(\hat{y}, y) := \|y - \hat{y}\|^2$$

Training as an optimization problem

- Given: Training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ with labels y_i.
- We specify an error measure D, and define the total error on the training set as

$$J(\mathbf{w}) := \sum_{i=1}^{n} D(f_{\mathbf{w}}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$$

Not examinable.

Training problem

In summary, neural network training attempts to solve the optimization problem

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w})$$

using gradient descent. For feed-forward networks, the gradient descent algorithm takes a specific form that is called *backpropagation*.

> Backpropagation is gradient descent applied to $J(\mathbf{w})$ in a feed-forward network.

In practice: Stochastic gradient descent

- The vector **w** can be very high-dimensional. In high dimensions, computing a gradient is computationally expensive, because we have to make "small changes" to **w** in many different directions and compare them to each other.
- Each time the gradient algorithm computes $J(\mathbf{w}')$ for a changed value $\mathbf{w}'$, we have to apply the network to every data point, since $J(\mathbf{w}') = \sum_{i=1}^{n} D(f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$.
- To save computation, the gradient algorithm typically computes $D(f_{\mathbf{w}'}(\tilde{\mathbf{x}}_i), \tilde{y}_i)$ only for some small subset of a the training data. This subset is called a *mini batch*, and the resulting algorithm is called **stochastic gradient descent**.

Not examinable.

BACKPROPAGATION

Neural network training optimization problem

$$\min_{\mathbf{w}} J(\mathbf{w})$$

The application of gradient descent to this problem is called *backpropagation*.

> Backpropagation is gradient descent applied to $J(\mathbf{w})$ in a feed-forward network.

Deriving backpropagation

- We have to evaluate the derivative $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

Not examinable.

Deriving backpropagation

- We have to evaluate the derivative $\nabla_{\mathbf{w}} J(\mathbf{w})$.
- Since J is additive over training points, $J(\mathbf{w}) = \sum_n J_n(\mathbf{w})$, it suffices to derive $\nabla_{\mathbf{w}} J_n(\mathbf{w})$.

Not examinable.

Chain Rule

Recall from calculus: Chain rule

Consider a composition of functions $f \circ g(x) = f(g(x))$.

$$\frac{d(f \circ g)}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

If the derivatives of f and g are f' and g', that means: $\frac{d(f \circ g)}{dx}(x) = f'(g(x))g'(x)$

Application to feed-forward network

Let $\mathbf{w}^{(k)}$ denote the weights in layer k. The function represented by the network is

$$f_{\mathbf{w}}(\mathbf{x}) \; = \; f_{\mathbf{w}}^{(K)} \circ \cdots \circ f_{\mathbf{w}}^{(1)}(\mathbf{x}) \; = \; f_{\mathbf{w}^{(K)}}^{(K)} \circ \cdots \circ f_{\mathbf{w}^{(1)}}^{(1)}(\mathbf{x})$$

To solve the optimization problem, we have to compute derivatives of the form

$$\frac{d}{d\mathbf{w}} D(f_{\mathbf{w}}(\mathbf{x}_n), y_n) = \frac{dD(\,\bullet\,, y_n)}{df_{\mathbf{w}}} \frac{df_{\mathbf{w}}}{d\mathbf{w}}$$

Not examinable.

Decomposing the Derivatives

- The chain rule means we compute the derivates layer by layer.
- Suppose we are only interested in the weights of layer k, and keep all other weights fixed. The function f represented by the network is then

$$f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\mathbf{w}^{(k)}} \circ f^{(k-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

- The first $k-1$ layers enter only as the function value of $\mathbf{x}$, so we define

$$\mathbf{z}^{(k)} := f^{(k-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

  and get

$$f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\mathbf{w}^{(k)}}(\mathbf{z}^{(k)})$$

- If we differentiate with respect to $\mathbf{w}^{(k)}$, the chain rule gives

$$\frac{d}{d\mathbf{w}^{(k)}} f_{\mathbf{w}^{(k)}}(\mathbf{x}) \;=\; \frac{df^{(K)}}{df^{(K-1)}} \cdots \frac{df^{(k+1)}}{df^{(k)}} \;\cdot\; \frac{df^{(k)}_{\mathbf{w}^{(k)}}}{d\mathbf{w}^{(k)}}$$

Not examinable.

WITHIN A SINGLE LAYER

- Each $f^{(k)}$ is a vector-valued function $f^{(k)} : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$.
- It is parametrized by the weights $\mathbf{w}^{(k)}$ of the *k*th layer and takes an input vector $\mathbf{z} \in \mathbb{R}^{d_k}$.
- We write $f^{(k)}(\mathbf{z}, \mathbf{w}^{(k)})$.

Layer-wise derivative

Since $f^{(k)}$ and $f^{(k-1)}$ are vector-valued, we get a Jacobian matrix

$$\frac{df^{(k+1)}}{df^{(k)}} = \begin{pmatrix} \frac{\partial f_1^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_1^{(k+1)}}{\partial f_{d_k}^{(k)}} \\ \vdots & & \vdots \\ \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_{d_k}^{(k)}} \end{pmatrix} =: \Delta^{(k)}(\mathbf{z}, \mathbf{w}^{(k+1)})$$

- $\Delta^{(k)}$ is a matrix of size $d_{k+1} \times d_k$.
- The derivatives in the matrix quantify how $f^{(k+1)}$ reacts to changes in the argument of $f^{(k)}$ if the weights $\mathbf{w}^{(k+1)}$ and $\mathbf{w}^{(k)}$ of both functions are fixed.

Not examinable.

Backpropagation algorithm

Let $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(K)}$ be the current settings of the layer weights. These have either been computed in the previous iteration, or (in the first iteration) are initialized at random.

Step 1: Forward pass

We start with an input vector $\mathbf{x}$ and compute

$$\mathbf{z}^{(k)} := f^{(k)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

for all layers k.

Step 2: Backward pass

- Start with the last layer. Update the weights $\mathbf{w}^{(K)}$ by performing a gradient step on

  $$D\big(f^{(K)}(\mathbf{z}^{(K)}, \mathbf{w}^{(K)}), y\big)$$

  regarded as a function of $\mathbf{w}^{(K)}$ (so $\mathbf{z}^{(K)}$ and y are fixed). Denote the updated weights $\tilde{\mathbf{w}}^{(K)}$.
- Move backwards one layer at a time. At layer k, we have already computed updates $\tilde{\mathbf{w}}^{(K)}, \dots, \tilde{\mathbf{w}}^{(k+1)}$. Update $\mathbf{w}^{(k)}$ by a gradient step, where the derivative is computed as

  $$\Delta^{(K-1)}(\mathbf{z}^{(K-1)}, \tilde{\mathbf{w}}^{(K)}) \cdot \ldots \cdot \Delta^{(k)}(\mathbf{z}^{(k)}, \tilde{\mathbf{w}}^{(k+1)}) \frac{df^{(k)}}{d\mathbf{w}^{(k)}}(\mathbf{z}, \mathbf{w}^{(k)})$$

On reaching level 1, go back to step 1 and recompute the $\mathbf{z}^{(k)}$ using the updated weights.

Not examinable.

SUMMARY: BACKPROPAGATION

- Backpropagation is a gradient descent method for the optimization problem

  $$\min_{\mathbf{w}} J(\mathbf{w}) = \sum_{i=1}^{N} D(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$

  D must be chosen such that it is additive over data points.
- It alternates between forward passes that update the layer-wise function values $\mathbf{z}^{(k)}$ given the current weights, and backward passes that update the weights using the current $\mathbf{z}^{(k)}$.
- The layered architecture means we can (1) compute each $\mathbf{z}^{(k)}$ from $\mathbf{z}^{(k-1)}$ and (2) we can use the weight updates computed in layers $K, \ldots, k+1$ to update weights in layer k.

Not examinable.

BACKGROUND: GRAPHS

The following slides summarize some background that may be helpful if you have not encountered the concept of a graph before.

UNDIRECTED GRAPHS

A **graph** consists of:

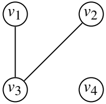
- A set of **vertices** $\mathcal{V} = \{v_1, \dots, v_n\}$. (The vertices are also called **nodes**.)
- A set $\mathcal{E}$ of **edges** between these vertices.
  Each edge is a pair (v, v'), where v and v' are vertices. If $(v, v') \in \mathcal{E}$, we think of v and v' as connected.

We depict a graph by representing each node as a circle, and connecting two nodes by a line if they share an edge.

Example

Here is the graph given by:

- The vertex set $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$.
- The edge set $\mathcal{E} = \{(v_1, v_3), (v_2, v_3)\}$.

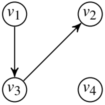


Note the edges in this graph have no direction: If we draw a graph in this manner, and replace the edge (v_1, v_3) by (v_3, v_1), the graph still looks the same.
A graph in which we do not distinguish between the edge (v, v') and the edge (v', v), for any two vertices v and v', is called **undirected**.

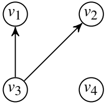
Directed Graphs

- If we regard (v, v') and (v', v) as two separate objects, each pair of vertices can be associated with two edges.
- If we distinguish the order of vertices within edges, the graph is called a **directed graph**.
- In other words: In an undirected graph, an edge (v, v') is a connection *between* the vertices v and v'. An a directed graph, (v, v') is a connection *from v to v'*.

We indicate that in the diagram by replacing the line between v to v' by an arrow from v to v':



$$\mathcal{E} = \{(v_1, v_3), (v_3, v_2)\}$$



$$\mathcal{E} = \{(v_3, v_1), (v_3, v_2)\}$$

In the class

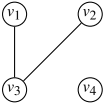
- Directed graphs are used in the transition diagrams of Markov chains, to represent directed graphical models (e.g. HMMs), and for neural networks. The "web graph" that represents (part of) the world wide web in the PageRank example is also a directed graph.
- Undirected graphs are used e.g. in Markov random fields.
- In Markov random fields, we also use weighted graphs (see below).

Adjacency Matrix

- A graph with n vertices $v_1, \ldots, v_n$ and edge set $\mathcal{E}$ can be represented as an $n \times n$-matrix A.
- The entries of this matrix are:

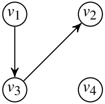
$$A_{ij} := \begin{cases} 1 & (v_i, v_j) \in \mathcal{E} \\ 0 & (v_i, v_j) \notin \mathcal{E} \end{cases} \qquad \text{for } i, j \in \{1, \ldots, n\}$$

- If the graph is directed, then $A_{ij} = A_{ji}$, so the matrix is symmetric.
- If the graph is undirected, A_{ij} and A_{ji} may be different (but do not have to be). The adjacency matrix of an undirected graph is not symmetric.
- Here are the adjacency matrices for the graphs on the previous two slides:



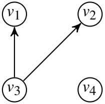
$\mathcal{E} = \{(v_1, v_3), (v_2, v_3)\}$

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$



$\mathcal{E} = \{(v_1, v_3), (v_3, v_2)\}$

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$



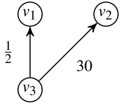
$\mathcal{E} = \{(v_3, v_1), (v_3, v_2)\}$

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

WEIGHTED GRAPH

- We can also define an adjacency matrix whose entries are arbitrary scalars, rather than only 0 or 1.
- In this case, we think of the entry A_{ij} as a *weight* of the edge (v_i, v_j).
- Such a graph is called a **weighted graph**.
- Note this definition is very general: We can think of every square matrix with real-valued entries as a weighted graph.
- If we draw a weighted graph, we write the edge weight next to the edge. If the weight is zero, we often omit drawing the edge.
- For example, here is a weighted graph with vertex set v_1, v_2, v_3:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{2} & 30 & 0 \end{pmatrix}$$



- We had encountered weighted, undirected graphs as neighborhood graphs of Markov random fields. Undirected again means $A_{ij} = A_{ji}$ for all vertices i and j.
- Another example (of a weighted graph that is directed, as the example above) would be a social network, where each vertex represents a user, and A_{ij} is the number of emails user i has sent to user j. In this case, the edge weights would be non-negative integers.