01.112 Machine Learning

SUTD, Fall 2018

Reuben R.W. Wang

Instructor: *Professor Liang Zheng* and *Wei Lu*.
Instructor office hours: *1200 - 1300 @1.409*.
Instructor email: `zheng_liang@sutd.edu.sg` \ `wei_lu@sutd.edu.sg`.

Contents

Chapter 1

Overview

Machine learning distinguishes itself from the conventional techniques of hardcoding used in traditional software applications. It is a branch of artificial intelligence which consists of tasks, performance of those tasks and experience. It is concerned with the design and development of algorithms that allows a computer to evolve behaviours based on empirical data. As intelligence requires knowledge, it is necessary for the computer to acquire such knowledge. The machine optimizes a performance criterion using sample data from past experiences. Machine learning is used when either, the necessary human expertise do not exist, humans are unable to find the underlying insight from large volumes of data or the solution needs to be constantly adapted.

§1.1 Introduction to Machine Learning

We start with a *model*, which the machine uses to predict an output from a inputs to this model. The model is trained/optimized during the *training phase* (phase of teaching the system how to learn) and its accuracy is tested during the *testing phase*. It is **important** to make the training and testing data similar so that the machine will perform optimally when implemented. Commonly, 'observable' data constitutes a *training set* which is used in training. The testing data however is unobserved and taken from the *universal set*. With the training data, we need find an appropriate mathematical model for our data and make good assumptions so that the tests will turn out favourably.

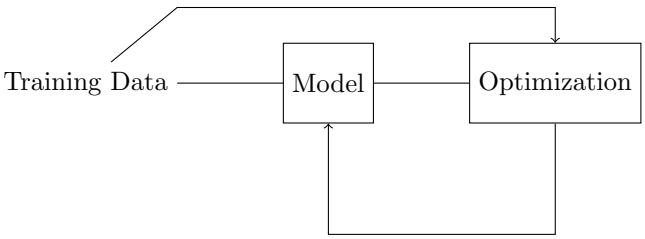

Figure 1.1: Rudimentary Machine Learning Training Model

Figure 1.2: Rudimentary Machine Learning Testing Model

§1.1.1 Performance

There are several factors that affect the performance of a machine's learning. These are:

- Quality of training data.
- Form and extent of initial background (prior) knowledge.
- Type of feedback.
- Learning algorithm used.
- **Modelling**.
- **Optimization**.

§1.2 Types of Machine Learning

§1.2.1 Supervised Learning

(Inputs: $\{x_n \in R^d, y_n \in R\}_{n=1}^N$)

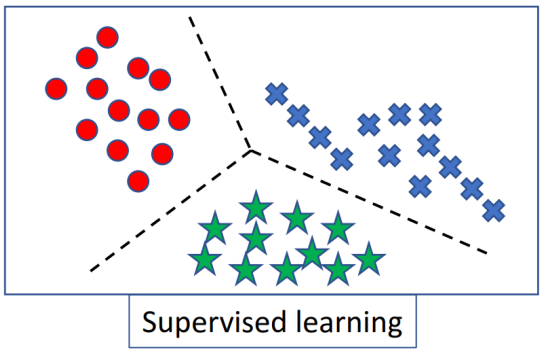

Figure 1.3: Visualization of Supervised Learning

The data set contains labels for each data point, denoted as the y_n entries. Hence the employed techniques are:

- Prediction (e.g. *Linear* and *non-linear regression* are forms of supervised learning, where choosing between linear and non-linear models is a hard problem).
- Classification (e.g. *Linear* and *non-linear classifiers* which partition data in the state space are also forms of supervised learning - used in email spam filters, fraud detection, etc).

§1.2.2 Unsupervised Learning

(Inputs: $\{x_n \in R^d\}_{n=1}^N$)

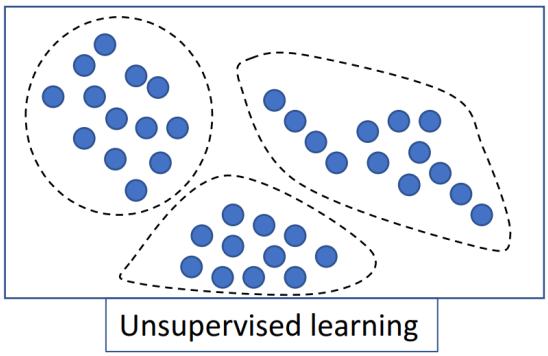

Figure 1.4: Visualization of Unsupervised Learning

The data set does **not** contains labels for each data point. Hence the employed techniques are:

- Clustering (data is grouped according to distinct features which constitute data subsets).
- Probability distribution estimation.
- Finding association.
- Dimensionality reduction / Subspace learning (utilizes mathematical projective methods to gain better insight of the data).

Semi-Supervised Learning

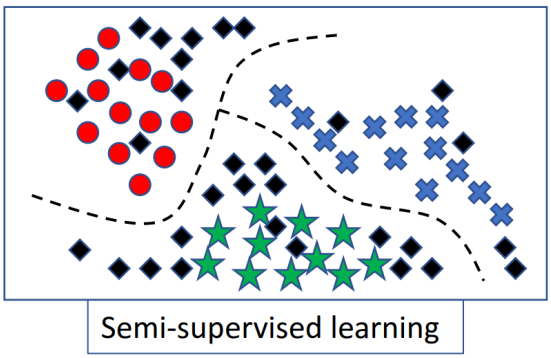

Figure 1.5: Visualization of Semi-Supervised Learning

A portion of the data has labels, whereas the remaining does not.

§1.2.3 Reinforcement Learning

It is a form of decision making (robot tasks, chess player machine, etc). It is a state of the art machine learning technique, and is based on learning by rewards from a sequence of actions.

Example

Consider a robot agent tasked with improving the state of the environment. The robot then takes some actions, after which feedback from the environment is fed to the robot in the form of the environments posterior state. Simultaneously, rewards are given to the robot based on the state of the environment. The robot then uses this feedback to train its actions and learn which actions are beneficial for improving the environment.

§1.2.4 Deep Learning

Deep learning attempts to simulate the workings of the human brain (neural architecture). It consists of a multi-layered structure/network which responds to different input 'stimuli' to produce the required outputs. Some applications are: *facial recognition, handwriting recognition, image captioning, video prediction, person re-identification in videos, object recognition and distance detection for self-driving cars, generative models, etc.*

Deep learning is largely considered as a black box, and many researchers are working to explain what goes on in this black box of innumerous parameters.

§ **Learning Outcomes** §

- Define machine learning in terms of algorithms, tasks, performance and experience.
- List four main types of machine learning.
- Describe some potential dangers in machine learning.

Chapter 2

Regression

Regression is a core class in machine learning, and a very classic machine learning task. In the problem of regression, the task is to fit a mathematical model to the training data. We assess its performance by calculating the prediction error (some distance measure between data points and model).

§2.1 Overview of Regression

Following the '*task, performance, experience*' paradigm of machine learning, we establish what each of these steps imply for regression.

Regression Paradigm:

- **Task**: to find/propose a function $f : \mathbb{R}^d \to \mathbb{R}$ such that the ground truth output is approximately equal to the predicted output. That is to say $y \approx f(x; \theta)$.
- **Experience**: (or data used in learning) comes from training data, which is a d dimensional data set of *features/vectors* $x = \{(x_1, x_2, ..., x_n)^{(1)}, ..., (x_1, x_2, ..., x_n)^{(d)}\}$.
- **Performance**: this is found by computing the *prediction error*, which is some function of $y - f(x; \theta)$.

Let's now delve a little deeper into these steps.

§2.2 Methodology

§2.2.1 Features

From our data set, we can extract parameters that will be used to characterized inputs. We call these *features* (or *feature vectors*). These are mathematically represented as d dimensional vectors $x \in \mathbb{R}^d$ where each entry denotes a parameter associated with characterization.

§2.2.2 Training vs Testing Data

We now look at 2 different subsets of data from the *universal data set*. For machine learning, we partition data into *training data sets* S_n and the *testing data sets* S_*.

Definition 2.2.1. Training Data:

$$S_n = \{(x^{(i)}, y^{(i)}) | i = 1, 2, ..., n\} \tag{2.1}$$

- *Where the* $x^{(i)} = (x_1^{(i)}, ..., x_d^{(i)})^T \in \mathbb{R}^d$ *are the features/inputs.*
- *Where the* $y^{(i)} \in \mathbb{R}$ *are responses/outputs.*

Definition 2.2.2. Linear Model: *The model (or hypothesis class)* $\mathcal{H}$ *is defined as a set of linear functions* $f : \mathbb{R}^d \to \mathbb{R}$*, described by*

$$f(x; \theta; \theta_0) = \theta_0 + \theta_1 x_1 + ... + \theta_d x_d = \theta_0 + \theta^T x \tag{2.2}$$

where $\theta \in \mathbb{R}^d$ *and* $\theta_0 \in \mathbb{R}$ *are known as the model parameters.*

In the above definition, each $f \in \mathcal{H}$ is what we call a *predictor* or *hypothesis*.

Definition 2.2.3. Training Loss/Objective: *We define a functional called the training loss* $\mathcal{L}(f; S_n)$ *on our training data from which we use this functional to find a predictor* $\hat{f}$ *that minimizes the objective function* $\mathcal{L}$*.*

$$\mathcal{L}(f; R_n) = \frac{1}{n} \sum_{(x,y) \in S_n} \frac{1}{2} \big(y - f(x)\big)^2 \tag{2.3}$$

Definition 2.2.4. Test Loss/Objective: *Similar to test loss, given a specific predictor* $\hat{f} \in \mathcal{H}$*, the test loss function* $\mathcal{R}(\hat{f}; R_*)$ *gives us a metric to determine how well* $\hat{f}$ *generalizes to new data. The test loss is defined as:*

$$\mathcal{R}(\hat{f}; R_*) = \frac{1}{n} \sum_{(x,y) \in S_*} \frac{1}{2} \big(y - \hat{f}(x)\big)^2 \tag{2.4}$$

The smaller the test loss, the better the performance of our model on the data.

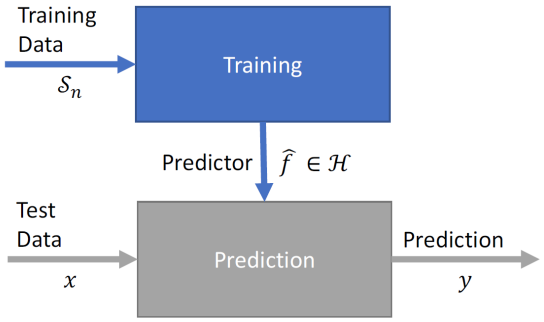

Figure 2.1: Training and Prediction Scheme

Be clear about the difference or relation between the training and test loss. The training loss is used to search for appropriate predictors whereas the test loss is used as a metric to assess the effectiveness of our found predictor after training. We can visualize the training and prediction scheme as in figure 2.1 above.

> **Note:** The training loss and test loss functions on the same predictor $\hat{f}$ can be quite different.

The goal of machine learning is to find a predictor $\hat{f} \in \mathcal{H}$ that **generalizes** well, which is to say that it predicts well on the test data set S_*.

> **Note:** A key assumption of training and prediction is that the test data and training data are *identically distributed*.

§2.2.3 Model Selection

When we are searching for a good model, there are 2 major issues that could arise. These are known as *underfitting* and *overfitting*. Hence, *model selection* is finding a model of the right size such that is fits well with both training and testing data.

1. **Underfitting** is when we use a model with too few parameters, causing a poor fit to the training data. Consequently, it will also not fit well with testing data and is an unfavourable model. *If the model $\mathcal{H}$ is too small, then $\hat{f} \in \mathcal{H}$ performs poorly on training data and poorly on testing data.*

2. **Overfitting** is when we use a model with too many parameters, causing a overly good fit to the training data. This causes the loss function to be higher in training data which is unfavourable. *If the model $\mathcal{H}$ is too big, then $\hat{f} \in \mathcal{H}$ performs well on training data but not testing data.*

Often, it is difficult to determine the number of parameters to use because the parameter space can be very large.

§2.3 Model Optimization

§2.3.1 Loss and Risk

We define the loss function as:

$$\boxed{\text{Loss}(z) = \frac{1}{2}z^2} \tag{2.5}$$

Where we square the error so as to penalizes big errors more heavily. The *empirical risk* otherwise known as *training loss* (as earlier defined) can be decomposed to give relevant terms. These are:

- **Point loss**: $\mathcal{L}_1(\theta; x, y) = \text{Loss}(y - f(x; \theta))$
- **Average loss**: $\mathcal{L}_n = \frac{1}{n}\sum_{(x,y)\in S_n} \mathcal{L}_1(\theta; x, y) = \frac{1}{n}\sum_{(x,y)\in S_n} \frac{1}{2}\big(y - f(x;\theta)\big)^2$

Over here, we have introduced the θ model parameters as a function argument because they sufficiently define our predictor function $\hat{f}$.

> **Note:** The subscripts on the $\mathcal{L}$ loss functions are **not** indices, but indicate the size of the training data set.

§2.3.2 Gradient

In any form of optimzation, it is essential that we compute the gradient of our object function. As such, recall the gradient of a multivariable function. In our context of a training loss objective function, its gradient with respect to the parameters θ would be:

$$\begin{aligned} \boldsymbol{\nabla}\mathcal{L}_n(\theta; S_n) &= \begin{bmatrix} \frac{\partial \mathcal{L}_n}{\partial \theta_1}(\theta; S_n) \\ \frac{\partial \mathcal{L}_n}{\partial \theta_2}(\theta; S_n) \\ \vdots \\ \frac{\partial \mathcal{L}_n}{\partial \theta_n}(\theta; S_n) \end{bmatrix} = \frac{1}{n}\sum_{(x,y)\in S_n} \begin{bmatrix} \frac{\partial \mathcal{L}_1}{\partial \theta_1}(\theta; S_n) \\ \frac{\partial \mathcal{L}_1}{\partial \theta_2}(\theta; S_n) \\ \vdots \\ \frac{\partial \mathcal{L}_1}{\partial \theta_n}(\theta; S_n) \end{bmatrix} \\ \Rightarrow\ \boldsymbol{\nabla}\mathcal{L}_n(\theta; S_n) &= \frac{1}{n}\sum_{(x,y)\in S_n} \boldsymbol{\nabla}\mathcal{L}_1(\theta; S_n) \end{aligned} \tag{2.6}$$

So we see that the training gradient is the **average** of the point gradients.

§2.3.3 Exact Solution and Gradient Descent

If there are **no** constraints on the parameters, then we simply set the gradient to 0 and solve for the parameters. We run through all the solutions to find the parameter that has the smallest training loss. If an analytical solution is not a viable option, then the goal is now to have an algorithm that decreases the value of the parameters $\mathcal{L}_n$ as we traverse the θ parameter space. Let's look at the *gradient descent algorithm* which accomplishes this.

Gradient Descent Algorithm:

- We first randomly initialize θ.
- We then follow the negative direction of the gradient, which is the direction that causes the loss function to decrease. $\theta \to \big(\theta - \eta_k \boldsymbol{\nabla} \mathcal{L}(\theta)\big)$, where η_k is the *learning rate* and k is the iteration index.
- We keep repeating step 2 until we converge to a minimum value.

We need to find an appropriate learning rate η_k that has high enough resolution not to miss the minimum point, but also not too small to cause the algorithm to be unnecessarily slow. When we apply the gradient descent algorithm, we check that our multivariable $\mathcal{L}(\theta)$ function is differentiable in the neighbourhood of θ, and decreases as we go away from θ in the $-\boldsymbol{\nabla}\mathcal{L}(\theta)$ direction. If this is true, then it follows that for $\theta_{k+1} = \theta_k - \eta \boldsymbol{\nabla} \mathcal{L}(\theta)$ for small enough η, we have that $\mathcal{L}(\theta_k) \geq \mathcal{L}(\theta_{k+1})$.

Note: Gradient descent leads us to a *local minimum*, which is **not necessarily** the global minimum. This means that choosing different starting points (θ_0) may lead to different local minima. Typically, we perform gradient descent from several starting points, and run through all the local minima to find the parameter that has the smallest training loss. However, if our loss function is *strictly convex*, the local minimum **is** the global minimum.

§2.3.4 Sub-Gradient

In the event that not the entire loss function on your parameter space is differentiable, we use what is called a *sub-gradient* to continue the gradient descent. The sub-gradient is any vector v such that for all x', we have

$$\boxed{f(x') - f(x) \geq v^T (x' - x)} \tag{2.7}$$

The premise is that your initial randomly chosen θ_0 has a local differentiable neighbourhood, however as we traverse along the hypersurface with the gradient descent algorithm, we reach a θ_i that has a non-differentiable neighbourhood. When this occurs, we use the sub-gradient method to prevent the algorithm from coming to an abrupt stop.

Note: The sub-gradient method is **not** a descent method, and the function value can increase in subsequent iterations.

§2.3.5 Stochastic Gradient Descent (SGD)

In practice however, the conventional gradient descent is **not** a very practical algorithm. This is because if we have a very large data set, we will have to calculate the average of the point gradients for every data point which will take an enormous amount of time. the trick is that we

estimate the gradient by averaging over a smaller randomly sampled *minibatch* (subset of the training data).

$$\begin{aligned}\boldsymbol{\nabla}\mathcal{L}_n(\theta; S_n) &\approx \boldsymbol{\nabla}\mathcal{L}_m(\theta; B_m) \\ &= \frac{1}{m}\sum_{(x,y)\in B_m} \boldsymbol{\nabla}\mathcal{L}_1(\theta; x, y), \quad \text{where } m < n\end{aligned} \tag{2.8}$$

Stochastic Gradient Descent Algorithm:

- Initialize θ randomly.
- Select a minibatch B_m of the data from S_n at random. We then perform the gradient descent via $\theta \to \theta - \eta_k \boldsymbol{\nabla}\mathcal{L}(\theta; B_m)$.
- Repeat step 2 until convergence.

As shown, steps for the SGD algorithm mirror closely the steps for the standard gradient descent algorithm. The difference that we are using less data than the entire training set provides.

To further improve the SGD, we can make adjustments to certain algorithm parameters. For instance, we can have the learning rate be a function of iteration steps so that it better ensures convergence at the minimum (e.g. $\eta_k = 0.9\eta_{k-1}$). Be careful that during implementation, you do not reduce the training rate so drastically per iteration such that it takes an exorbitant amount of time to converge. We can also define a *momentum* which helps to reduce fluctuations of the gradient while working with a data minibatch. The update thus look as such:

$$\theta^{(k+1)} \to \theta^{(k)} - \eta_k \Delta^{(k)} \tag{2.9}$$

where $\Delta^{(k)} = (1-\epsilon)\Delta^{(k-1)} + \epsilon \boldsymbol{\nabla}\mathcal{L}_m(\theta; B_m)$.

§2.4 Multivariate Linear Regression

§2.4.1 Method of Least Squares

Given a set of input/output data $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(n)}, y^{(n)})\}$ with $x^{(i)} \in \mathbb{R}^d$, $y^{(i)} \in \mathbb{R}$. We use a **linear** model which is called linear because of the form of the predictor function $f(x; \theta, \theta_0) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_d x_d$ with $\theta \in \mathbb{R}^d$, $\theta_0 \in \mathbb{R}$. The method of least squares then employs a loss function similar to what we have already seen:

$$\begin{aligned}\mathcal{L}_1(\theta, \theta_0; x, y) &= \frac{1}{2}\big(y - (\theta^T x + \theta_0)\big)^2 \\ \mathcal{L}_n(\theta, \theta_0; x, y) &= \frac{1}{n}\sum_{(x,y)\in S_n} \mathcal{L}_1(\theta, \theta_0; x, y)\end{aligned} \tag{2.10}$$

But notice here our model of the predictor is *affine* in the features, and not purely linear. To simplify this model, we can use a trick called the *constant feature trick*.

Constant Feature Trick

To modify the predictor into a purely linear function, we simply increase the dimension of the feature space by one.

$$(x \in \mathbb{R}^d) \to (\tilde{x} \in \mathbb{R}^{d+1}) \tag{2.11}$$

So our inputs and outputs are now $\{(\tilde{x}^{(1)}, y^{(1)}), (\tilde{x}^{(2)}, y^{(2)}), ..., (\tilde{x}^{(n)}, y^{(n)})\}$ which makes our predictor model $f(\tilde{\theta}; \tilde{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_d x_d = \tilde{\theta}^T \tilde{x}$ where $x_0 = 1$. Our model is now purely linear. Often, we will drop the tilde notation and assume the use of the constant feature trick.

Least Square Analysis

For our point loss function, the point gradient is computed to be

$$\boldsymbol{\nabla} \mathcal{L}_1(\theta; x, y) = \begin{bmatrix} -x_1(y - \theta^T x) \\ -x_2(y - \theta^T x) \\ \vdots \\ -x_d(y - \theta^T x) \end{bmatrix} = - \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} (y - \theta^T x) = -x(y - \theta^T x) \tag{2.12}$$

We can also employ a matrix representation to make our expressions a little more elegant. Let us first define the following for clarity:

$$X = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(n)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & ... & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & ... & x_d^{(2)} \\ \vdots & & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & ... & x_d^{(n)} \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \tag{2.13}$$

With this, we can reformulate our training gradient into a a simple equation as such:

$$\begin{aligned} \boldsymbol{\nabla} \mathcal{L}_n(\theta; x, y) &= \frac{1}{n} \sum_{(x,y) \in S_n} -x(y - \theta^T x) \\ &= \frac{1}{n} \sum_{(x,y) \in S_n} (-xy + x\theta^T x) \\ &= \frac{1}{n} \sum_{(x,y) \in S_n} (-xy + xx^T \theta) \\ &= (-\frac{1}{n} X^T Y + \frac{1}{n} X^T X \theta) \equiv -B + A\theta \end{aligned} \tag{2.14}$$

So each iteration of the gradient descent algorithm becomes:

$$\boxed{\theta \to \theta - \eta_k (\frac{1}{n} X^T Y - \frac{1}{n} X^T X \theta)} \tag{2.15}$$

To find an exact solution to this problem, let us work under some implicit assumptions and then express them explicitly after our analysis. First, given that our training loss is convex, we take

the global minimum as satisfying $\nabla \mathcal{L}_n(\hat{\theta}) = 0$. From the matrix expression we derived above and the necessary condition for minima, we get

$$\begin{aligned} & \frac{1}{n} X^T Y = \frac{1}{n} X^T X \hat{\theta} \\ \Rightarrow \quad & \boxed{\hat{\theta} = (X^T X)^{-1} X^T Y} \end{aligned} \tag{2.16}$$

We immediately we see that above in above analytical solution, we require that $X^T X$ be an *invertible* matrix. This implies that we require $n \geq d$ (number of data point more than or equal to the size of the feature vectors). Also if there is an immense number of features (d very large), it would very computationally heavy to invert the matrix (best algorithm achieves a complexity of $\mathcal{O}(n^{2.376})$). In that case, it is practical to use the SGD instead.

§2.5 Regularization

Often times, unbounded optimization of parameters may lead to the parameters scaling to extreme values. To curb this, we can impose a regularizer (much like a Lagrangian multiplier) to limit the unmitigated scaling of our parameters.

§2.5.1 Ridge Regression

Consider a model with the linear predictor $y \approx \theta_0 + \theta_1 x_2 + ... + \theta_d x_d$. How do we ensure that $\theta_k = 0$ when x_k is irrelevant? We add a *regularizer* so that our objective function becomes:

$$\mathcal{L}_{n,\lambda}(\theta) = \frac{1}{n} \sum_{(x,y) \in S_n} \frac{1}{2}(y - \theta^T x)^2 + \frac{\lambda}{2} \|\theta\|^2 \tag{2.17}$$

where λ is the regularizer which puts '*pressure*' to simplify the model (penalizes any increase in the parameter values). Why this is called **ridge** regression is because in the method of least squares, it is often that along certain parameter degrees of freedom on the hypersurface, we have a long flat hyperline similar to a ridge. This additional regularizer term helps to break the linearity of this ridge for minima search.

For the unconstrained optimization of a convex loss function, the exact solution can once again be found as done in the previous section as follows:

$$\begin{aligned} & \nabla \mathcal{L}_{n,\lambda}(\hat{\theta}) = 0 \\ \Rightarrow \quad & \frac{1}{n} X^T Y = \frac{1}{n} X^T X \hat{\theta} + \lambda \hat{\theta} \\ \Rightarrow \quad & \boxed{\hat{\theta} = (n\lambda \mathbb{I} + X^T X)^{-1} X^T Y} \end{aligned} \tag{2.18}$$

With regularization, the descent algorithm iterations are as such:

$$\boxed{\theta \to (1 - \eta_k \lambda)\theta - \eta_k (\frac{1}{n} X^T Y - \frac{1}{n} X^T X \theta)} \tag{2.19}$$

Note: Regularizers are used **only** during the training phase and applied to the training loss but **not** the training error. This is so because it should only be used as a means to better 'tweak' the predictor function but is not a modification to the error metric.

A visualization of how training and test errors could typically vary by adding a regularizer is shown below (figure 2.2).

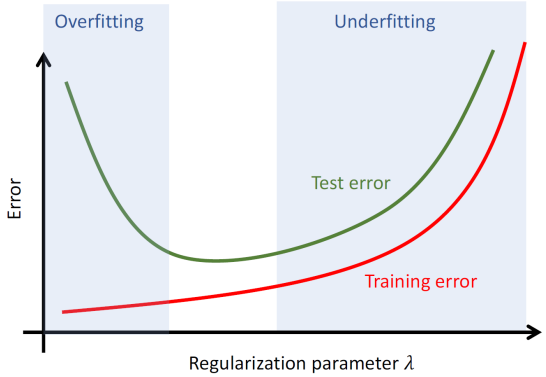

Figure 2.2: Error vs Regularization Parameter Plot

From figure 2.2, we can see that there is an optimal space of values for which to set the regularizer which will achieve the lowest error.

§2.6 Hyperparameters

As seen in the previous section, the regularization parameter λ is an example of a *hyperparameter*. Hyperparameters are parameters whose values are set **before** the learning process begins. They affect the complexity of the model. The question then arises, how do we pick these hyperparameters if we do not generally have access to test data? The solution is to create a *validation set*.

§2.6.1 Validation Set

With the introduction of hyperparameters, we now split our data into 3 subsets. These are:

- **Test Set (S_*):** This is used for evaluating and reporting performance at the end.
- **Training Set (S_n):** This is used to determine the optimal parameters in a model during the training phase.
- **Validation Set ($S_{\mathbf{val}}$):** This is used for model selection and acts as a proxy for the test set.

Just as the training and testing sets had metrics to tune and ascertain the model's effectiveness, we can also define a *validation loss* function that will essentially have the same purpose as the test loss objective function.

Example:

For ridge regression, our validation loss function is defined as:

$$\mathcal{R}(\hat{\theta}; S_{\text{val}}) = \frac{1}{n} \sum_{(x,y) \in S_{\text{val}}} \frac{1}{2}(y - \hat{\theta}^T x)^2 \tag{2.20}$$

The validation essentially creates an added layer of deterministic testing from an observable data set. We can think of this as an extension of the training and model picking phase, where we are further optimizing our model (now with added complexity from the hyperparameter). A visualization of this is shown below (figure 2.3).

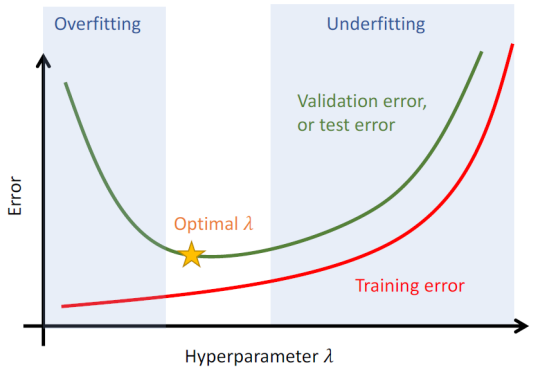

Figure 2.3: Model Selection with a Hyperparameter

Chapter 3

Classification

For most of the pedagogical purposes of this class, we consider a 2-class classification paradigm. This is the simplest case. Classification is a close cousin of regression, where instead of finding a function that best models for the expected continuous output value of our data, we aim to group our outputs into discrete classes.

§3.1 Overview of Classification

Similar to regression, *classification* follows the '*task, performance, experience*' paradigm of machine learning. The result space assumes a binary classification problem $\{-1, +1\}$ as mentioned above (e.g. alive or dead, cancer vs. not cancer).

Classifier Paradigm:

- **Task**: We need to find a function/*classifier* $h : \mathbb{R}^d \to \{-1, +1\}$ such that the ground truth output is approximately equal to the predicted output $y = h(x; \theta)$ (note that the mapping is onto a **discrete** space).
- **Experience**: (or data used in learning) comes from training data, which is a d-dimensional data set of *features/vectors* $x = \{(x_1, x_2, ..., x_n)^{(1)}, ..., (x_1, x_2, ..., x_n)^{(d)}\}$.
- **Performance** is found by computing the *prediction error*, $y - h(x; \theta)$ (note that the error is also discrete, for which it is either correct or incorrect).

§3.2 Linear Classification

Very similar to regression, we also have a training data set for classifiers.

Definition 3.2.1. Training Data:

$$S_n = \{(x^{(i)}, y^{(i)}) | i = 1, 2, ..., n\} \tag{3.1}$$

- *Where the* $x^{(i)} = (x_1^{(i)}, ..., x_d^{(i)})^T \in \mathbb{R}^d$ *are the features/inputs.*
- *Where the* $y^{(i)} \in \{-1, +1\}$ *are the responses/outputs.*

The model of a *linear classifier* is then given by: A set of linear classifiers $h : \mathbb{R}^d \to \{-1, +1\}$, defined by

$$\begin{aligned} h(x; \theta, \theta_0) &= \text{sign}(\theta_d x_d + ... + \theta_1 x_1 + \theta_0) \\ &= \text{sign}(\theta^T x + \theta_0) \end{aligned} \tag{3.2}$$

where the 'sign' function is defined by:

$$\text{sign}(z) = \begin{cases} +1, & z \geq 0 \\ -1, & z < 0 \end{cases} \tag{3.3}$$

Note: Some people define $z(0) = 0$ but we do not adopt that in this class.

The goal here (during training) is also to solve for the model parameters $\theta \in \mathbb{R}^d$ and $\theta_0 \in \mathbb{R}$.

Definition 3.2.2. Test Loss: *The point and average loss functions for test data are defined as:*

$$\mathcal{R}_1(\theta, \theta_0; x, y) = [[y \neq h(x; \theta, \theta_0]] \tag{3.4}$$

$$\mathcal{R}(\theta, \theta_0; S_*) = \frac{1}{n} \sum_{(x,y) \in S_*} \mathcal{R}_1(\theta, \theta_0; x, y) \tag{3.5}$$

where $[[.]]$ *is the* ***indicator function*** *that returns* 1 *if its argument is true, and* 0 *otherwise.*

§3.2.1 Decision Regions

A classifier h partitions the space into *decision regions* that is separated by *decision boundaries*. All the points in each region map to the same label. For linear classifiers, these regions are known as *half spaces*.

Note: Several different regions could have the same label.

§3.2.2 Decision Boundaries

For linear classifiers, the decision boundary is a *hyperplane* of dimension $d-1$ given by the equation

$$\boxed{\theta^T x + \theta_0 = 0} \tag{3.6}$$

where vector θ is orthogonal to the decision boundary and points in the direction of region labelled $+1$. It then follows that if for some data point x_0, having $\theta^T x + \theta_0 > 0$ implies an output of $+1$.

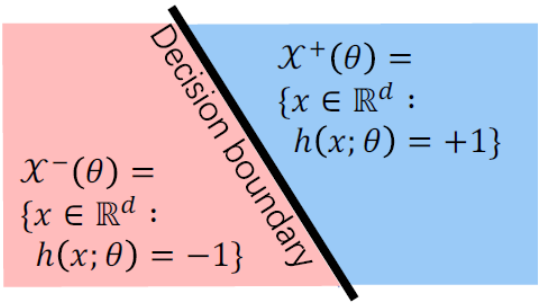

Figure 3.1: Visualization of Decision Boundaries in a Linear Classifier

§3.2.3 Linearly Separable

An important property we will be looking at for some of the more basic classification algorithms to come is known as *linear separability*.

Definition 3.2.3. Linearly Separable: *The training data S_n is linearly separable if there exists a parameter θ and θ_0 such that for all $(x, y) \in S_n$, we have*

$$y(\theta^T x + \theta_0) > 0 \tag{3.7}$$

§3.2.4 Constant Feature Trick

This is exactly the same as the constant feature trick we did for regression where we increase the dimension of our x feature by 1 ($d \to d+1$) and renamed it $\tilde{x}$ with a constant 1 entry at the end. The new model is then:

$$x \to \tilde{x} \in \mathbb{R}^{d+1}, \quad \theta \to \tilde{\theta} \in \mathbb{R}^{d+1} \tag{3.8}$$

$$\Rightarrow \quad h(x;\theta,\theta_0) = \text{sign}(\tilde{\theta}^T \tilde{x}) \tag{3.9}$$

and the test loss is then

$$\mathcal{R}_1(\tilde{\theta};\tilde{x},y) = [[y \neq h(\tilde{x},\tilde{\theta})]] \tag{3.10}$$

$$\mathcal{R}_n = \frac{1}{n} \sum_{(x,y)\in S_*} \mathcal{R}_1(\tilde{\theta};\tilde{x},y) \tag{3.11}$$

§3.3 The Perceptron Algorithm

Linear classifiers are also often referred to as *perceptrons*. These were designed to resemble *neurons*. The perceptron model is illustrated below.

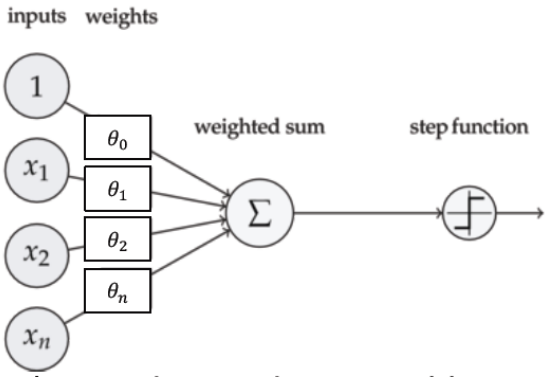

Figure 3.2: Perceptron Model

§3.3.1 Zero-One Loss

We begin by first exploring the simplest instance of a loss function for classifiers. Here, we are trying to define a function that behaves as follows:

$$\mathcal{L}_1(\theta; x, y) = \begin{cases} 1, & y \neq h(x;\theta) \\ 0, & \text{otherwise} \end{cases} \tag{3.12}$$

Above, the 1 outcome refers to a misclassification (point loss increases) and the 0 outcome indicates that the data point is on the boundary. To construct such a loss function, We first note that

1. $y(\theta^T x) \leq 0$ if $(\theta^T x)$ and (y) differ in sign ($y(\theta^T x) \leq 0 \Rightarrow$ misclassification).
2. $\theta^T x = 0$ implies that the data point is on the boundary.

As such, we can construct our loss function as:

$$\mathcal{L}_1(\theta; x, y) = [[y(\theta^T x) \geq 0]] = \text{Loss}\big(y(\theta^T x)\big) \tag{3.13}$$

where $\text{Loss}(z) = [[z \geq 0]]$ is the zero-one loss with $z = y(\theta^T x)$.

Definition 3.3.1. Training Loss:

$$\text{Loss}(z) = [[z \leq 0]] \tag{3.14}$$

$$\mathcal{L}_1(\theta; x, y) = \text{Loss}\big(y(\theta^T x)\big) \tag{3.15}$$

$$\mathcal{L}_n(\theta; S_n) = \frac{1}{n} \sum_{(x,y)\in S_n} \mathcal{L}_1(\theta; x, y) \tag{3.16}$$

Because zero-one loss is discrete, we cannot use the gradient descent algorithm. Instead, we use the *mistake-driven algorithm* in replacement.

Mistake-Driven Algorithm:

1. Initialize $\theta = 0$
2. For each data point $(x, y) \in S_n$
   (a) Check if $h(x; \theta) = y$
   (b) If not, update θ to correct the mistake
3. Repeat Step (2) until no mistakes are found.

A specific implementation of the mistake-driven algorithm is known as the perceptron algorithm. This is given below:

Perceptron Algorithm:

1. Initialize $\theta = 0$
2. For each data point $(x, y) \in S_n$, if $y(\theta^T x) \leq 0$, then we update $\theta \leftarrow \theta + xy$.
3. Repeat Step (2) until no mistakes are found.

Note: For the perceptron algorithm, notice that we initialize the θ parameter vector as a vector of zeros and not an arbitrary point like in SGD.

1. Training Set (Linearly Separable)
$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$$
2. Model (Set of Perceptrons)
$$h(x; \theta) = \text{sign}(\theta_1 x_1 + \cdots + \theta_d x_d)$$
3. Training Loss (Fraction of Misclassified/Boundary Points)
$$\mathcal{L}_n(\theta) = \frac{1}{n} \sum_{(x,y) \in S_n} [\![y(\theta^\top x) \leq 0]\!]$$
3. Algorithm (Mistake-Driven Algorithm)

Figure 3.3: Perceptron Algorithm Summary

§3.4 Hinge Loss

Using the zero-one loss is **not** a very useful tool on real data sets as it causes the model to fail if there exists data that do not satisfy the linearly separable condition. This makes it non-robust on real data sets. As such, we look for an alternate loss functions for classification. A better alternative is known as *hinge loss*.

$$\text{Loss}_H(z) = \max\{1 - z, 0\}, \quad z = y(\theta^T x) \tag{3.17}$$

$$\mathcal{L}_n(\theta; x, y) = \frac{1}{n} \sum_{\text{data}(x,y)} \text{Loss}(z) \tag{3.18}$$

This loss definition penalizes large mistakes, and accounts for '*near-mistakes*' in cases like $0 \leq z \leq 1$. A visualization of the hinge-loss function is given in figure 3.4 below.

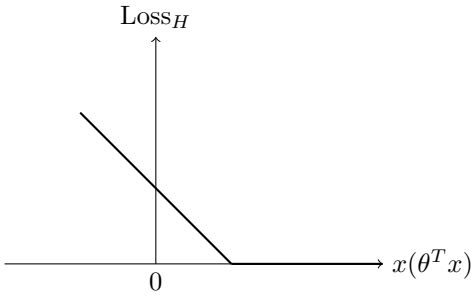

Figure 3.4: Hinge Loss Function

Because of the differentiable nature of the hinge loss function, we can find its gradient!

$$\begin{aligned} \mathcal{L}_n &= \frac{1}{n} \sum_{(x,y)\in S_n} \text{Loss}(y(\theta^T x)) \\ &= \frac{1}{n} \sum_{(x,y)\in S_n} \max\{1 - y(\theta^T x), 0\} \end{aligned} \tag{3.19}$$

The gradient is then (using the chain rule):

$$\nabla_\theta \text{Loss}(y(\theta^T x)) = \begin{cases} 0, & y(\theta^T x) > 1 \\ -yx, & \text{otherwise} \end{cases} \tag{3.20}$$

Now that we have a gradient, we can apply the stochastic gradient descent (SGD) algorithm with hinge loss as follows:

Hinge Loss Algorithm:

1. Initialize $\theta = 0$
2. Randomly select $(x, y) \in S_n$, if $y(\theta^T x) \leq 1$, then we update $\theta \to \theta + \eta_k xy$.
3. Repeat Step (2) until convergence.

Note: The hinge loss algorithm differs from perceptron algorithm in the following ways:

- We check if $z \leq 1$ rather than $z \leq 0$.
- We have a decreasing η_k for each iteration instead of the constant $\eta = 1$.
- We select data at random rather than in sequence.

1. Training Set (Not Necessarily Linearly Separable)
$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})$$
2. Model (Set of Perceptrons)
$$h(x; \theta) = \text{sign}(\theta_1 x_1 + \cdots + \theta_d x_d)$$
3. Training Loss (Hinge Loss)
$$\mathcal{L}_n(\theta) = \frac{1}{n} \sum_{(x,y) \in \mathcal{S}_n} \max\{1 - y(\theta^\top x), 0\}$$
3. Algorithm (Gradient Descent)
$$\theta \longleftarrow \theta + \frac{\eta_k}{n} \sum_{(x,y) \in \mathcal{S}_n} yx$$

Figure 3.5: Hinge Loss Algorithm Summary

§3.5 Logistic Regression

§3.5.1 Probabilistic Model

To further improve on our classification model, we can attempt to model the conditional probability that the label $y = +1$ given a feature x. This is done by using a common statistical model known as the *sigmoid function*.

$$\begin{aligned} &h : \mathbb{R}^d \to [0, 1] \\ &h(x; \theta) = \mathbb{P}(y = +1 | x) = \text{sigmoid}(\theta^T x) \end{aligned} \tag{3.21}$$

§3.5.2 Sigmoid Function

Also known as the logistic function, the *sigmoid function* is the go-to for modelling the continuous probability (also known as confidence) that a label y matches a feature x. A visualization of the sigmoid function is shown below.

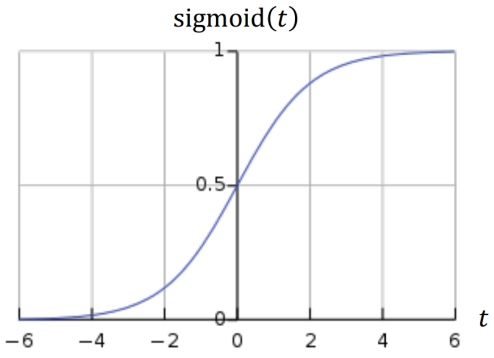

Figure 3.6: Sigmoid Function

Mathematically, the sigmoid function is defined as:

$$\text{sigmoid} : \mathbb{R} \to [0, 1] \tag{3.22}$$

$$\text{sigmoid}(t) = \frac{1}{1 + e^{-t}} \tag{3.23}$$

Additionally, a useful relation between equivalent positive/negative values of t is as follows:

$$\begin{aligned} \text{sigmoid}(-t) &= \frac{1}{1 + e^{t}} \\ &= \frac{e^{-}t}{e^{-}t + 1} \\ &= 1 - \frac{1}{e^{-}t + 1} = 1 - \text{sigmoid}(t) \end{aligned} \tag{3.24}$$

§3.5.3 Sigmoid Neurons

To improve the perceptron neuron model, they introduced the *sigmoid neuron* model as illustrated in the figure below.

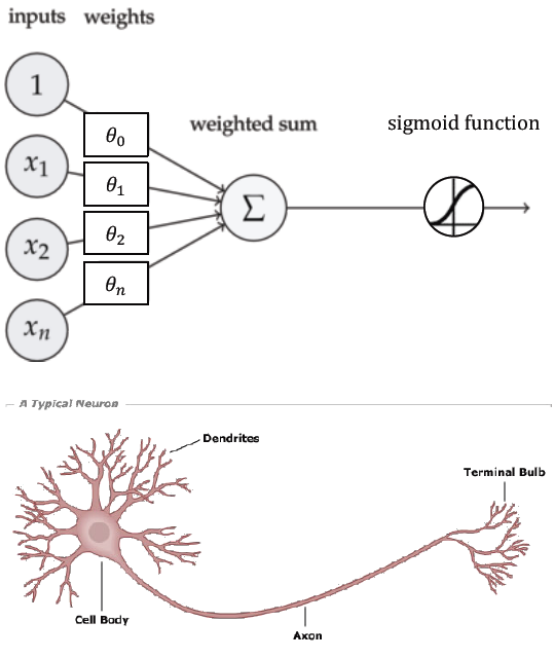

Figure 3.7: Sigmoid Neuron Model

§3.5.4 Label Probabilities

Utilizing the sigmoid function in our context, we look to find the conditional probability of attaining a $y = \pm 1$ classification given some data point x.

$$\mathbb{P}(y = +1|x) = \text{sigmoid}(\theta^T x) = \text{sigmoid}\big(y(\theta^T x)\big) \tag{3.25}$$

$$\mathbb{P}(y = -1|x) = \text{sigmoid}(-\theta^T x) = \text{sigmoid}\big(y(\theta^T x)\big) \tag{3.26}$$

$$\Rightarrow \ \mathbb{P}(y|x) = \text{sigmoid}\big(y(\theta^T x)\big), \quad y \in \{-1, +1\} \tag{3.27}$$

§3.5.5 Label Predictions

How do we now predict these probability quantities in relation to which label is more probable? From our conventions, we have the inequality:

$$\mathbb{P}(y = +1|x) \geq \mathbb{P}(y = -1|x) \iff h(x;\theta) \geq \frac{1}{2} \tag{3.28}$$

With this, we can see that we interpret the value of our classifier function as such:

- $h(x;\theta) \geq 1/2$ implies that we predict this x data point's label is $y = +1$.
- $h(x;\theta) < 1/2$ implies that we predict this x data point's label is $y = -1$.

Recall that the decision boundary is defined by $\theta^T x = 0$ (dropping the tilde notation). As such, we have the following equivalences:

- $h(x;\theta) \geq 1/2 \iff \text{sigmoid}(\theta^T x) \geq 1/2 \iff \theta^T x \geq 0$
- $h(x;\theta) < 1/2 \iff \text{sigmoid}(\theta^T x) < 1/2 \iff \theta^T x < 0$

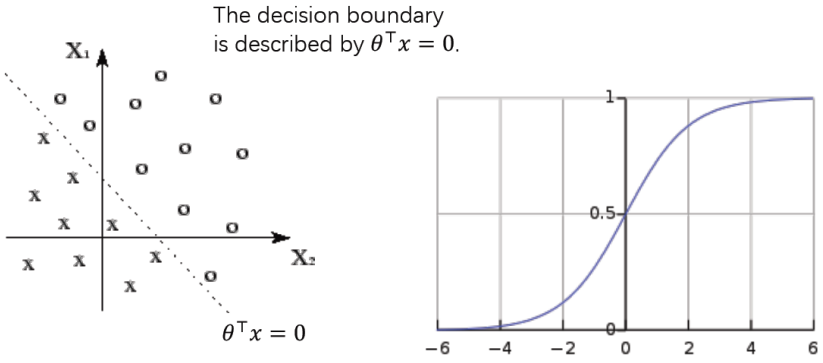

Figure 3.8: Sigmoid Function Decision Boundary Visualization

§3.5.6 Likelihood

Given a training set S_n, we can define what is known as the likelihood function L.

$$\begin{aligned} L(\theta; S_n) &= \mathbb{P}(y^{(1)}, ..., y^{(n)}|x^{(1)}, ..., x^{(n)}) \\ &= \mathbb{P}(y^{(1)}|x^{(1)})...\mathbb{P}(x^{(n)}|y^{(n)}) = \prod_{(x,y)\in S_n} \mathbb{P}(y|x) \end{aligned} \tag{3.29}$$

Above, we have assumed that the data points are all independent. The goal is then to **maximize** this likelihood function since each point's conditional probability represents the probability of attaining a correctly assigned label. Notice that the following optimization problems are equivalent:

- $\max\{L(\theta; S_n)\}$
- $\max\{\log L(\theta; S_n)\}$
- $\min\{-\frac{1}{n}\log L(\theta; S_n)\}$

As such, we define the *logistic loss* function by the last equivalent quantity above.

Definition 3.5.1. Logistic Loss:

$$\begin{aligned}\mathcal{L}_n(\theta; S_n) &= -\frac{1}{n}\log L(\theta; S_n) \\ &= -\frac{1}{n}\sum_{(x,y)\in S_n}\log \mathbb{P}(y|x) \\ &= -\frac{1}{n}\sum_{(x,y)\in S_n}\log \frac{1}{1+e^{-y(\theta^T x)}} \\ &= \frac{1}{n}\sum_{(x,y)\in S_n}\log\Big(1+e^{-y(\theta^T x)}\Big) \\ &= \frac{1}{n}\sum_{(x,y)\in S_n}\text{Loss}\big(y(\theta^T x)\big)\end{aligned} \tag{3.30}$$

As a little summary, we list all the training loss functions below.

§ **Recap of Training Losses** §

The training loss is given by:

$$\mathcal{L}_n(\theta; S_n) = \frac{1}{n}\sum_{(x,y)\in S_n}\text{Loss}(z), \quad z = y(\theta^T x) \tag{3.31}$$

The respective point losses are given by:

- **Zero-One Point Loss**: $\text{Loss}_{01} = [[z \leq 0]]$
- **Hinge Point Loss**: $\text{Loss}_{\text{H}} = \max\{1-z, 0\}$
- **Logistic Point Loss**: $\text{Loss}_{\text{L}} = \log(1+e^{-z})$

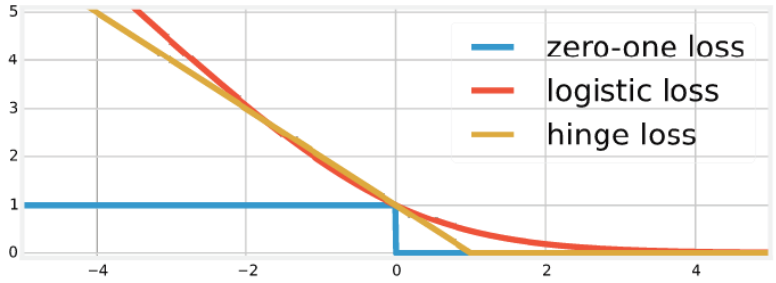

Figure 3.9: Visualization of Different Point Loss Functions

§3.5.7 Logistic Gradient

Of course, we also compute the gradient of the logistic loss function with respect to its parameters θ to minimize it.

$$\begin{aligned}\nabla_\theta \text{Loss}_\text{L}\big(y(\theta^T x)\big) &= \begin{cases} x(\text{sigmoid}(\theta^T x) - 1), & y = +1 \\ x(\text{sigmoid}(\theta^T x) - 0), & y = -1 \end{cases} \\ &= x(\text{sigmoid}(\theta^T x) - [[y = +1]])\end{aligned} \tag{3.32}$$

So we see that the training gradient is given by:

$$\nabla \mathcal{L}_n(\theta; S_n) = \frac{1}{n} \sum_{(x,y)\in S_n} x \left(\frac{1}{1 + \exp\big(-y(\theta^T x)\big)} - [[y = +1]] \right) \tag{3.33}$$

Logistic Gradient Descent Algorithm:

1. Initialize $\theta = 0$
2. Update $\theta \to \theta - \frac{\eta_k}{n} \sum_{(x,y)\in S_n} x(\text{sigmoid}(\theta^T x) - [[y = 1]])$.
3. Repeat Step (2) until convergence.

1. Training Set (Not Necessarily Linearly Separable)
$$\big(x^{(1)}, y^{(1)}\big), \big(x^{(2)}, y^{(2)}\big), \ldots, \big(x^{(n)}, y^{(n)}\big)$$
2. Model (Set of Sigmoid Neurons)
$$h(x; \theta) = \text{sigmoid}(\theta_1 x_1 + \cdots + \theta_d x_d)$$
3. Training Loss (Logistic Loss)
$$\mathcal{L}_n(\theta) = \frac{1}{n} \sum_{(x,y)\in S_n} \log\big(1 + e^{-y(\theta^\top x)}\big)$$
3. Algorithm (Gradient Descent)
$$\theta \longleftarrow \theta - \frac{\eta_k}{n} \sum_{(x,y)\in S_n} x(h(x; \theta) - [\![y = 1]\!])$$

Figure 3.10: Logistic Regression Algorithm Summary

Note: We need to distinguish between discriminative and generative models in classification. Discriminative models are trying to learn conditional distributions $\mathbb{P}(X|Y)$ whereas generative models are learning joint distributions $\mathbb{P}(X, Y)$.

Chapter 4

Clustering

Clustering belongs under the category of unsupervised learning. Recall that in unsupervised learning, we do not have any labels for our feature vectors. Some useful applications of unsupervised learning are data compression (image compression via pixel RGB partitioning and quantization), improvement of classification/regression, etc. Clustering is the process of partitioning a set of data into a set of (hopefully) meaningful sub-classes, called clusters.

§4.1 What is Clustering?

Definition 4.1.1. Clusters: *Collection of data points that are "similar" to one another and collectively should be treated as a group. As a collection, data points differ sufficiently between clusters.*

The clustering problem is formally presented as follows:

Clustering Problem:

Given the input training data $S_n = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, where each $x^{(i)} \in \mathbb{R}^d$, we want to produce an output of clusters (pseudo-labels) $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_k \subset \{1, 2, ..., n\}$ such that every data point is in one and **only** one cluster.

There are several ways to specify a cluster:

- Explicitly listing all its elements.
- Using a representative of its elements (e.g. centroid, exemplar, etc).

§4.2 Basic Clustering Methodology

There are 2 main approaches to clustering.

1. **Agglomerative**: Pairs of items and smaller clusters are slowly linked together to form larger clusters.
2. **Divisive**: Items are initially placed in one cluster and successively divided into separate groups.

§4.2.1 Agglomerative Single-Link

A *single-link* connects all points that are within a *threshold distance* together. Before we get to the algorithm, we define a quantity known as the *intercluster distance* as the distance between the closest 2 points of 2 given clusters. The algorithm for doing this is as follows:

Algorithm:

- Start by first setting all n data points to be in their own cluster.
- Merge the 2 closest clusters, where close-ness is defined by the intercluster distance.
- Repeat step 2 $n - k$ times to get k clusters.

§4.2.2 Agglomerative Complete-Link (Clique)

A *complete-link* has the implication that all points in a cluster **must** be within a certain threshold distance. In the *threshold distance matrix*, a clique is a **complete graph** (every pair of graph vertices are connected by an edge). Agglomerative complete-link algorithms then aim to find the maximal clique of any chosen point.

§4.3 Methods of Characterizing Clusters

There are different techniques we can employ to characterize clusters in our data set. Some of these are listed and elaborated on below.

§4.3.1 Classes

The label applied by a clustering algorithm. These labels could be *hard* or *fuzzy* labels.

- **Hard label**: Either is or is not a member of a cluster.
- **Fuzzy label**: Membership to a cluster is determined with probability.

§4.3.2 Distance/Similarity Measure

Definition 4.3.1. Distance Measure: *A value indicating how similar 2 data points are. For example, a common distance measure is the Euclidean distance between data points* $dist(x, y) = \|x - y\|$.

Note: Distance measures are also often referred to as *loss functions.*

Definition 4.3.2. Similarity Measure: *A value that quantifies the similarity between 2 data points. A common similarity measure is the cosine similarity* $\cos(x, y) = \frac{x^T y}{\|x\|\|y\|}$.

Note: Similarity measures are often referred to as *kernel functions.*

In distance calculation, we utilize feature vectors. Distance measures **must** satisfy the following properties:

- They should be based on feature values
- For all objects x_i, x_j, they have to satisfy $\text{dist}(x_i, x_j) \geq 0$ and $\text{dist}(x_i, x_j) = \text{dist}(x_j, x_i)$.
- For any object x_i, $\text{dist}(x_i, x_i) = 0$.
- $\text{dist}(x_i, x_j) \leq \text{dist}(x_i, x_k) + \text{dist}(x_k, x_j)$.

Some explicit forms of distance measures are given below:

- **Euclidean Distance**: $\sqrt{\sum_{f=1}^{|\text{feature}|}(x_{i,f} - x_{j,f})^2}$
- **Manhattan Distance**: $\sum_{f=1}^{|\text{feature}|} |(x_{i,f} - x_{j,f})|$
- **Minkowski (p) Distance**: $\sqrt[p]{\sum_{f=1}^{|\text{feature}|}(x_{i,f} - x_{j,f})^p}$
- **Mahalanobis Distance**: $(x_i - x_j)\boldsymbol{\nabla}^{-1}(x_i - x_j)^T$, where $\boldsymbol{\nabla}^{-1}$ is the covariance matrix of the data.
- **Mutual Neighbour Distance (MND)**: based on the number of nearest neighbours count.

We can also construct *distance* and *similarity* matrices. These are symmetric matrices where the (i, j) entries are the distance/similarity measures ($d_{i,j} \in \mathbb{R}$ or $s_{i,j} \in \{0, 1\}$) between items i and j. The diagonal is all 0's (distance) or all 1's (similarity).

$$D = \begin{bmatrix} 0 & d_{1,2} & ... & d_{1,n} \\ d_{2,1} & 0 & ... & d_{2,n} \\ \vdots & & \ddots & \vdots \\ d_{n,1} & d_{n,2} & ... & 0 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & s_{1,2} & ... & s_{1,n} \\ s_{2,1} & 1 & ... & s_{2,n} \\ \vdots & & \ddots & \vdots \\ s_{n,1} & s_{n,2} & ... & 1 \end{bmatrix} \tag{4.1}$$

Where for the entries above, $\forall i, j \in \{1, 2, ..., n\}$, we have $d_{i,j} = d_{j,i}$ and $s_{i,j} = s_{j,i}$. This matrix can also be visualized as a *weighted/undirected graph.* Each item is represented by a node, and the edge weight/edges represent the distance/similarity between 2 items (nodes).

Definition 4.3.3. Clustering Training Loss: *We define the clustering training loss as the sum of squared distance measures (or simply distances) to the closest representative. We have 3 possible training loss function forms that allow for optimization over representatives*

(cluster centers), clusters and both representatives and clusters respectively.

$$\mathcal{L}_{n,k}(z^{(1)},...,z^{(k)};S_n) = \sum_{i=1}^{k} \min_{1\leq j\leq k} \left\|x^{(i)} - z^{(j)}\right\|^2 \tag{4.2}$$

$$\mathcal{L}_{n,k}(\mathcal{C}_1,...,\mathcal{C}_k;S_n) = \sum_{j=1}^{n} \sum_{i\in\mathcal{C}_j} \min_{1\leq j\leq k} \left\|x^{(i)} - \frac{1}{|\mathcal{C}_j|} \sum_{i\in\rfloor_j} x^{(i')}\right\|^2 \tag{4.3}$$

$$\mathcal{L}_{n,k}(\mathcal{C}_1,...,\mathcal{C}_k;z^{(1)},...,z^{(k)};S_n) = \sum_{j=1}^{k} \sum_{i\in\mathcal{C}_j} \left\|x^{(i)} - z^{(j)}\right\|^2 \tag{4.4}$$

These objective functions above are too complex and not often used in practice.

§4.3.3 Deterministic vs Stochastic

§4.3.4 Hierarchical

In the hierarchical method, data points are connected into clusters using a point hierarchical structure. This is based on some methods of representing a hierarchy of data points. One example of the hierarchical method is the *hierarchical dendogram*. The hierarchical agglomerative is as follows:

Hierarchical Agglomerative Clustering:

1. Compute the distance matrix.
2. Put each data point in its own cluster.
3. Find the most similar pair of clusters by:
   (a) Merging pairs of clusters.
   (b) Updating the proximity matrix.
   (c) Repeating until all patterns in one cluster.

A visualization of hierarchical clustering is given in figure below.

	T1	T2	T3	T4	T5	T6	T7
T2	7						
T3	16	8					
T4	15	12	18				
T5	14	3	6	6			
T6	14	18	16	18	6		
T7	9	6	0	6	9	2	
T8	7	17	8	9	3	16	3

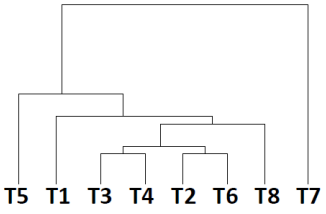

Figure 4.1: Hierarchical Dendograms

§4.3.5 Checking Clustering Validity

An important but difficult question to solve is 'how good are the clusters produced by a certain algorithm'? In other words, are the clusters produced by your algorithm valid for the intended purpose? Formulating an objective measure for this is non-trivial to say the least. Some approaches to validity checks are:

- External Assessment: Comparing clusters to a priori clusters.
- Internal Assessment: Determine if the clustering is intrinsically appropriate for the given data.
- Relative Assessment: Compare the results of several clustering methods with the same data set.

However, even before approaching the validity problem, it is first good to check some basic considerations:

- Data Preparation: Setting up the data to apply your clustering algorithm on (e.g. extraction, normalization).
- Similarity/Distance Measures: Which construction of distance measure is appropriate for our problem.
- Use of Domain Knowledge: Prior knowledge can affect/influence data preparation.
- Efficiency: How do we ensure that clusters are generated in a reasonable amount of time.

Example:

Voronoi diagrams are an example of clusterning. In a Voronoi diagram, we partition all the points in the space into regions according to their closest representative.

§4.4 K-Means

k-means clustering is a method of *vector quantization* and works by partitioning the input data set of n observations (data points) into k clusters such that each data point belongs to the cluster with its *nearest mean*, serving as a *representative/prototype* of the cluster. This results in a partitioning of the data space into what are known as *Voronoi cells* as mentioned earlier. In order to achieve this, we can once again utilize several optimization (training loss minimization $\min\{\mathcal{L}(x, y)\}$) algorithms. We will utilize the *coordinate descent* algorithm for k-means clustering.

Coordinate Descent Algorithm:

1. Find optimal x while holding y constant (finding the best cluster given centroids).
2. Find optimal y while holding x constant (finding the best centroid given clusters).
3. Repeat until convergence.

Note: Even though coordinate descent ensures that the training loss always decreases in each iteration, we may not arrive at the global minimum when the algorithm terminates.

We now look at the k-means algorithm which utilizes this notion of coordinate descent. The algorithm is as follows:

k-Means Algorithm:

1. *Initialization*: For the first iteration, we initialize the centroids $\{z^{(1)}, ..., z^{(k)}\}$ from the data (pick k data points as our cluster centers/cluster representative).
2. *Forming Clusters*: We then look through all $x^{(i)}$ points and assign them such that we have:

$$\mathcal{C}_j = \{i | x^{(i)} \text{ is closest to } z^{(j)}\} \tag{4.5}$$

   where $j \in \{1, ..., k\}$. That is to say, we run through all the data points and assign each of them to a cluster by looking for their corresponding nearest cluster representative.
3. *Computing Cluster Centers*: With these clusters, we then recompute the cluster centers $z^{(j)}$ by taking the point closest to the mean of that cluster. The mean is computed as follows:

$$z^{(j)} = \frac{1}{|\mathcal{C}_j|} \sum_{i \in \mathcal{C}_j} x^{(i)} \tag{4.6}$$

4. *Iteration*: Repeat steps 2 and 3 until the clusters and cluster centers no longer vary (convergence).

A visualization of what the k-means algorithm is doing in each iteration is given in figure 4.2 below.

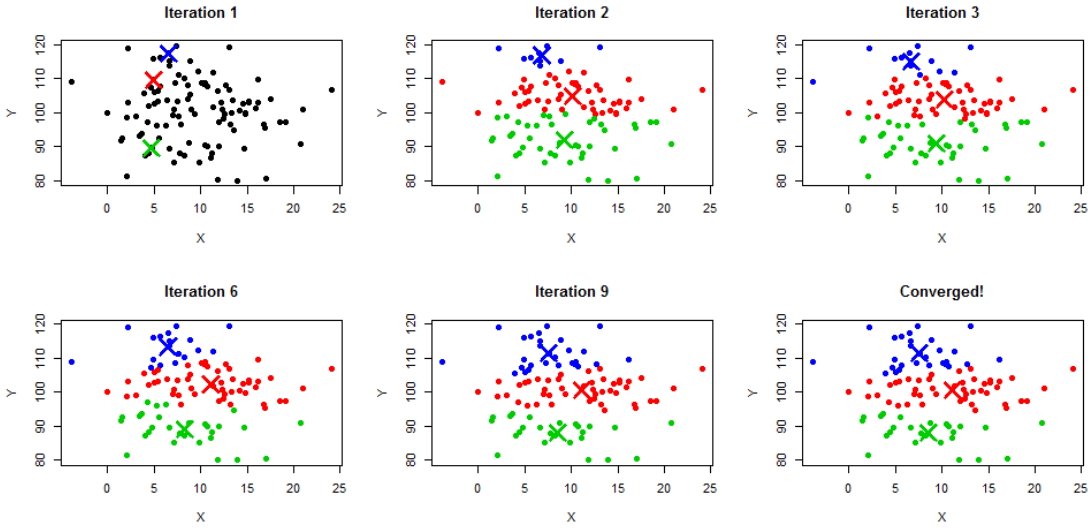

Figure 4.2: k-Means Clustering Iterations

§4.4.1 Initialization Issues

Some problems can arise with the use of k-means clustering. To avoid some of these, people have proposed the following solutions to their respective problems:

- What if we pick an initial point (centroid) that gives us an empty cluster (all data points fall outside of the Voronoi cell)? This is resolved by picking initial points that are already data points.
- What if we encounter bad local minima (much larger than the global minimum)? We can initialize the algorithm many times with different initial centroids, then compare and find the initialization with the smallest training loss.

§4.4.2 Choosing K

Another question to ask when using k-means clustering is 'what is the optimal number of clusters to have with a given data set?'. There are 2 common approaches to tackling this:

- Elbow Method
- Semi-Supervised Method

Chapter 5

Recommendation

Recommendation is usually viewed as a stand alone research topic in the field of machine learning and is not part of the classic machine learning paradigm (recommendation is more associated to areas such as data mining). An example of an application of recommendation is when Netflix saw a need to recommend relevant movies that their customers might be interested in based on past views. So the idea of recommendation is predicting the type of object that will be preferable to a user (e.g. missing data completion).

§5.1 Collaborative Filtering

Let's start by defining the general recommendation problem (the recommendation problem is also known as *matrix/tensor completion problem*). We have n users and m objects to be offered to the users. This can be represented by an $(n \times m)$ matrix Y. Each row is then the preference rating by each user for each movie (Y_{ij} = rating of object j by user i). However, our effective data set is even smaller because not every user will rate every object or even rate at all (matrix is *incomplete*)! The aim is then to *complete* the matrix and fill up all the entries.

> **Note:** This problem is close to a regression problem because the outputs have semantic relationships unlike in classification.

Collaborative filtering implies the notion of *cross-users*, which points out that users who share similar interests on items in the past are more likely to hold similar opinions on other items compared to randomly chosen users. There are different types of recommendation learning:

- Model Based: Here, we treat Y_{ai} as the responses. Given training data of the form $\{(a, i), Y_{ai}\}$, we want to predict a function $f : \{1, ..., n\} \times \{1, ..., m\} \to \mathbb{R}$.
- Memory Based: Here, we treat Y_{ai} as features. Given incomplete user ratings $Y_a \in \mathbb{R}^m$, we find structure in the data to predict missing values.

§5.2 K Nearest Neighbours

k nearest neighbours is a memory based learning algorithm and the main idea is to find k users $b_1, ..., b_k$ that are similar (neighbours) to user a. So we are going to use information from users $b_1, ..., b_k$ to predict ratings of user a. To do this, we need some kind of similarity measure in order to find a's k nearest neighbours. A useful similarity measure is known as the *correlation coefficient*, which is similar to cosine similarity as seen before.

Definition 5.2.1. Correlation Coefficient: *A numerical measure of the statistical relationship between two variables. For 2 vectors x and y, the correlation coefficient between x and y is given by:*

$$\text{corr}(x, y) = \cos(x - \bar{x}, y - \bar{y}) = \frac{(x - \bar{x})^T (y - \bar{y})}{\|x - \bar{x}\| \|y - \bar{y}\|} \tag{5.1}$$

where $\bar{x}$ and $\bar{y}$ are the averages of the entries of x and y.

Note: When we use correlation coefficients, we are subtracting away the mean value which in practice means that we are subtracting away the *user bias*.

§5.2.1 User Similarity

To compute the similarity between users a and b, we

1. Find $CR(a, b)$.
2. Let Z_a and Z_b be the vector of ratings in $CR(a, b)$ for each user.
3. We then compute the correlation coefficient between Z_a and Z_b

$$\text{sim}(a, b) = \text{corr}(Z_a, Z_b) \in [-1, +1] \tag{5.2}$$

$CR(a, b)$ stands for *common ratings* between user a and b, which is the set of objects rated by both a and b.

§5.2.2 Weighted Prediction

Let $\bar{Y}_b$ denote the average of object ratings by user b. To predict the rating Y_{ai} of user a for object i, we:

1. Ranks users b who have rated object i according to the value of $|\text{sim}(a, b)|$.
2. Let $kNN(a, i)$ be the set of k users with the highest similarity.
3. Let $(Y_{ai} - \bar{Y}_a)$ be the weighted average of $\pm(Y_{bi} - \bar{Y}_b), b \in kNN(a, i)$.
4. Let weight for $\pm(Y_{bi} - \bar{Y}_b)$ be proportional to $|\text{sim}(a, b)|$.

So from these steps, we get the formula:

$$\boxed{\hat{Y}_{ai} - \bar{Y}_a = \frac{\sum_{b \in kNN(a,i)} \text{sim}(a, b)(Y_{bi} - \bar{Y}_b)}{\sum_{b \in kNN(a,i)} |\text{sim}(a, b)|}} \tag{5.3}$$

Note: The above formula is **not** sensitive to the bias (mean) of each user, but **is** sensitive to the spread (variance). Also, there is no training loss and no training algorithm for kNN.

§5.3 Subspace Learning and Matrix Factorization

§5.3.1 Subspace Learning

The main idea of *subspace learning* is that the current space of our learning is largely redundant (we do not need such large dimentionality for our parameter space). Let's say we have completed rating vectors $\hat{Y}_1, ..., \hat{Y}_n \in \mathbb{R}^m$ which lie in some k-dimensional subspace $k << m$ (the *rank* is generally much smaller than m and n). This means that we can find just k linearly independent vectors to comprehensively represent $\hat{Y}_a$.

$$\hat{Y}_a = u_{a1}V_1 + u_{a2}V_2 + ... + u_{ak}V_k \tag{5.4}$$

§5.3.2 Matrix Factorization

With $V_j \in \mathbb{R}^m$ and the coefficients $u_{aj} \in \mathbb{R}$. As such, we can effectively decompose our matrix into an $(n \times k)$ matrix U of coefficients u_{ij} and an $(m \times k)$ matrix V where its columns are the k rating 'basis' vectors. This is known as *matrix factorization*. The reason we do this is to find a **complete** matrix of lower rank $\hat{Y}$ that is 'closest' to the incomplete matrix Y. $\hat{Y}$ is called the *low-rank approximation* of Y.

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1k} \\ u_{21} & u_{22} & \dots & u_{2k} \\ \vdots & & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nk} \end{bmatrix}, \quad V = \begin{bmatrix} \vdots & \vdots & & \vdots \\ V_1 & V_2 & \dots & V_k \\ \vdots & \vdots & & \vdots \end{bmatrix} \tag{5.5}$$

Note: When we talk about how close the matrices are, we are only referring to non-empty entries in Y to be used for comparison to $\hat{Y}$.

With this, we can write our low-rank approximation of Y as:

$$\boxed{\hat{Y} = UV^T} \tag{5.6}$$

Note: Although we get a **lower rank** matrix $\hat{Y}$, the dimension of the $\hat{Y}$ matrix is **still** $(n \times m)$ and **not** $(k \times k)$.

Now, it is good to look to an analogy to have a more tangible intuition of what is going here.

Analogy:

$\hat{Y}_{ai} = U_a^T V_i$. What does this mean and how do we interprete this? Let's say the objects we are dealing with here are movies. When watching movies, user a may have preferences on some genres (action, romance, etc) which constitute k factors. So we have that $U_a \in \mathbb{R}^k$ characterizes user a's preference on these k factors. Also, for some movie i, it can also be accurately described by these k genres. Hence $V_i \in \mathbb{R}^k$ describes how the k factors distribute in movie i. So, when we do the multiplication $U_a^T V_i$, we obtain how much user a likes movie i.

§5.3.3 Prediction

For an unknown rating Y_{ai} of user a for movie i, we predict

$$\hat{Y}_{ai} = (UV^T)_{ai} = U_a(V^T)_i \tag{5.7}$$

where U_a denotes the a-th row of matrix U and V_i denotes the i-row of matrix V. We also call these the *factor vectors* of user a and object i. Now that we have a prediction matrix $\hat{Y}$, we can construct a training loss function once again to optimize! Remember that for a loss function, we can only compute the loss against existing and observable data points, so we would have $< n \times m$ data points since our data matrix is incomplete. As such, we construct our training loss function as follows.

Definition 5.3.1. Recommendation Training Loss: *We define the recommendation training loss as the sum of squared differences between the entries of $\hat{Y}$ and Y along with regularizer terms to penalize unobserved data entries.*

$$\mathcal{L}_{n,k,\lambda}(U, V; Y) = \sum_{(a,i)\in D} \frac{1}{2}(\hat{Y}_{ai} - (UV^T)_{ai})^2 + \frac{\lambda}{2}(\|U\|^2 + \|V\|^2) \tag{5.8}$$

where D is the set of all (a, i) such that Y_{ai} is observed (has collected data).

In the definition, recall that U and V are matrices, so the norm that we are taking here is known as the *Frobenius norm.*

Definition 5.3.2. Frobenius Norm: *Given an $(n \times m)$ matrix A, the Frobenius norm of A is defined as:*

$$\|A\| = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij}^2} \tag{5.9}$$

Also, it is important to add the regularizer terms in this training loss because data entries in the low-rank approximation $\hat{Y}$ which do not have a corresponding entry in the actual Y matrix will not show up in the loss function. Hence, these entries could be scaled up to arbitrarily large values if they are not kept in check in some way (i.e. via the regularizers).

§5.3.4 Optimizing Training Loss

For recommendation learning, we have 2 matrix factors U and V that we need to account for. Hence, it is natural that we lean toward using the coordinate descent algorithm in order to find the minimum for our training loss function. To refresh your memory and also frame it for our recommendation learning context, the coordinate descent algorithm is as follows:

Alternating Least Squares Algorithm:

1. Initialize V by randomly generating its rows $V_1, V_2, ..., V_m \in \mathbb{R}^k$.
2. Fix the V elements and minimize the training loss function with respect to matrix U ($\min_U\{\mathcal{L}_{n,k,\lambda}(U, V; Y)\}$), i.e. for each user a, find U_a that minimizes
$$\sum_{i:(a,i)\in D} \frac{1}{2}(Y_{ai} - U_a^T V_i)^2 + \frac{\lambda}{2}\|U_a\|^2 \tag{5.10}$$
3. Fix the matrix U and minimize the training loss function with respect to matrix V ($\min_V\{\mathcal{L}_{n,k,\lambda}(U, V; Y)\}$), i.e. for each object i, find V_i that minimizes
$$\sum_{a:(a,i)\in D} \frac{1}{2}(Y_{ai} - U_a^T V_i)^2 + \frac{\lambda}{2}\|V_i\|^2 \tag{5.11}$$
4. Repeat steps 2 and 3 until convergence.

Additional, it would be good to think about the following considerations during implementation of this algorithm. How do we account for biases in our data? Can we simply subtract the average ratings of each user? Like k-means, the algorithm generally only converges to a local minimum, it would be good to perform multiple initializations and pick best result. Lastly, it would make our model more robust if we use a *validation set* (just as we did for regression) to pick the right hyperparameters k and λ.

§5.3.5 Validation Set

As mentioned above, since we are making use of hyperparameters, we would now like to employ the use a a validation set. So we now have the following data sets:

- **Test Set (S_*):** This is used for evaluating and reporting performance at the end.
- **Training Set (S_n):** This is used to determine the optimal parameters in a model during the training phase.
- **Validation Set ($S_{\mathbf{val}}$):** This is used for model selection and acts as a proxy for the test set.

Chapter 6

Support Vector Machines

A support vector machine (SVM) is a linear classifier which finds an optimal partitioning solution by maximizing the distance between the hyperplane decision boundaries and 'difficult points' close to these boundaries. SVM is a supervised learning model. The idea here is that if there are no points near the decision surface, then there are few uncertain classification decisions. The SVM aims to maximize the ***margin*** *around the separating hyperplane. If the decision boundary is a hypersurface, the margin is a hypervolume whose outer hypersurfaces run parallel to the boundary. Colloquially, we are trying to make the fattest hypervolume. The closest points to the decision boundary are used to define the margin surfaces and are called the* ***support vectors****. Solving SVMs is a quadratic programming problem and is seen by many as the most successful text classification method as of today.*

§6.1 Prerequisite Mathematics

First we look at some of the mathematics required to gain a comprehensive understanding of support vector machines. This is largely to do with the method of Lagrangian multipliers as we will be looking to solve a constrained optimization problem.

§6.1.1 Lagrangian Multipliers

In optimization, when we want to include constraints into our objective function, we utilize Lagrangian multipliers to do so. Let's say we are given some *primal objective function* $f(x)$ to optimize and it is subject to several constraints. There are 2 ways to approach such an optimization problem that we will be exploring, both of which utilize Lagrangian multipliers:

1. **Dual Problem**: This is where we set up a dual optimization problem where the constraints are 'nicer', and it is easier to implement a gradient descent algorithm. A more rigorous treatment of this topic can be found in chapter A of the appendix.
2. **Exact Solution**: This is an analytical method whereby we analytically solve the Lagrangian system of equations.

The new objective function built from the primal problem which incorporates the constraints and Lagrangian multipliers is known as the *Lagrangian*.

§6.1.2 Equality Constraints

Here, we explore formulating the *Lagrangian* with constraints that only have equality. Given an objective function $f(x)$ subject to l constraints $h_j(x) = 0$, the Lagrangian is given by:

$$\boxed{L(x, \lambda) = f(x) + \lambda_1 h_1(x) + ... + \lambda_l h_l(x) = f(x) + \sum_{j=1}^{l} \lambda_j h_j(x)} \tag{6.1}$$

§6.1.3 The Dual Paradigm

In the dual paradigm, there are 2 ways to approach solving this optimization problem. These are referred to as either playing the *primal game* or the *dual game*.

1. **The Primal Game**: Here, we first enforce the constraints and the aim to find x that **minimizes** $f(x)$.
$$\boxed{p^* = \min_x \left\{ \max_\lambda \{L(x, \lambda)\} \right\}} \tag{6.2}$$

2. **The Dual Game**: Here, we first compute the $l(\lambda) = \min_x\{L(x, \lambda)\}$ functions for each λ and then find the λ that **maximizes** $l(\lambda)$.
$$\boxed{d^* = \max_\lambda \left\{ \min_x \{L(x, \lambda)\} \right\}} \tag{6.3}$$

The primal problem is *lower bounded* by the dual problem ($p^* = \min_x\{\max_\lambda\{L(x, \lambda)\}\} \geq d^* = \max_\lambda\{\min_x\{L(x, \lambda)\}\}$). If $p^* = d^*$, then we say that we can exactly solve the primal problem by solving the dual problem.

§6.1.4 The Analytical Approach

Now if we want to have an exact analytical solution, we can utilize our knowledge of multivariable calculus and solve a system of equations using the necessary conditions for optimization.

Analytical Approach:

1. Write down the Lagrangian $L(x, \lambda) = f(x) + \sum_{j=1}^{l} \lambda_j h_j(x)$.
2. Solve for the critical points w.r.t x and λ:
$$\nabla_x L(x, \lambda) = 0, \quad \nabla_\lambda L(x, \lambda) = 0 \tag{6.4}$$
3. Pick the critical point which gives global minimum.

§6.1.5 Inequality Constraints

When our constraints consists of inequalities, this changes our problem slightly but thankfully the steps we use to optimize our objective function remains relatively unchanged. Given an objective function $f(x)$ subject to m constraints $g_j(x) \leq 0$, the Lagrangian is given by:

$$\boxed{L(x, \alpha) = f(x) + \alpha_1 g_1(x) + ... + \alpha_l g_m(x) = f(x) + \sum_{j=1}^{m} \alpha_j g_j(x)} \tag{6.5}$$

Inequalities require us to adopt additional necessary conditions to ensure that we are in fact at the critical points. These are known as the *Karush-Kuhn-Tucker* (KKT) conditions and *complimentary slackness*. The steps for an analytical solution are given below:

Analytical Approach:

1. $\nabla_x L(x, \lambda) = 0$
2. $g_j(x) \leq 0$ for all $j \in [1, m]$
3. $\alpha_j \geq 0$ for all $j \in [1, m]$
4. $\alpha_j g_j(x) = 0$ for all $j \in [1, m]$

§6.2 Computing Margins

As said in the introductory paragraph, our goal is to maximize the width of the margin around our decision boundary.

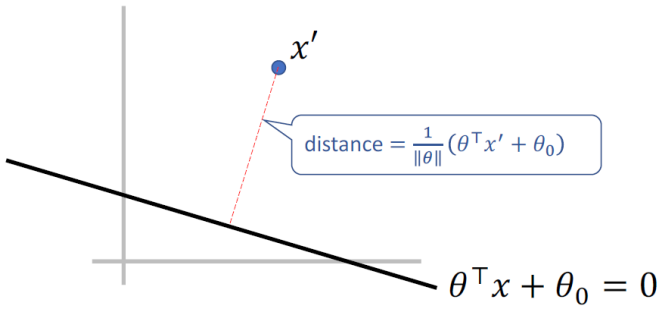

Figure 6.1: The Distance from the Decision Boundary

As such, our goal is to maximize $1/\|\theta\|$ which is half the *size of the decision boundary* (size of decision boundary is the normal distance between the 2 decision margin hyperplanes) subject to the constraint $y(\tilde{\theta}^T \tilde{x}) \geq 1$ for all data (x, y) where $y \in \{-1, +1\}$, for which the parity indicates which class the data point x falls into. Equivalently, we can minimize $\frac{1}{2}\|\theta\|^2$ subject to the constraint $y(\tilde{\theta}^T \tilde{x}) \geq 1$ for all data (x, y).

Definition 6.2.1. Support Vector: *Given a set of predicted parameters* $\{\hat{\theta}, \hat{\theta}_0\}$ *that define the decision boundary, the support vector is any vector that satisfies the equality:*

$$(\hat{\theta}^T x + \hat{\theta}_0)y = 1 \tag{6.6}$$

where x *is the feature vector and* y *is the response/label attached to that feature.*

So from here, the Lagrangian for our minimization problem is:

$$L(\theta, \alpha) = \frac{1}{2}\|\theta\|^2 + \sum_{(x,y)} \alpha_{x,y}\big(1 - y(\tilde{\theta}^T \tilde{x})\big) \tag{6.7}$$

Applying the KKT condition, we finally get to the optimization problem:

$$\max_{\alpha} \left\{ l(\alpha) = \sum_{(x,y)} \alpha_{x,y} - \frac{1}{2} \sum_{(x,y)} \sum_{(x',y')} \alpha_{x,y}\alpha_{x,'y'} yy'(\tilde{x}^T \tilde{x}') \right\} \tag{6.8}$$

Subject to $\alpha_{x,y} \geq 0$ for all (x, y) (complimentary slackness). The complimentary slackness condition ensures that only the support vectors affect the shifting of the decision boundary (**only for support vectors** do we have non-trivial/non-zero $\alpha_{x,y}$ values).

Note: Notice that for the dual problem above, we do not require the feature vectors themselves but only the value of their dot products $(\tilde{x}^T \tilde{x}')$. As such we can apply what is known as the *kernel trick* to simplify the computation in an actual implementation.

§6.3 SVM with Errors

There are cases where the linearly separable assumption could be violated. In this case, we have to change our primal problem by adding a *slack variable*. The primal problem then becomes:

$$\text{Objective Function}: \quad \min_{\theta} \left\{ \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{n} \sum_{(x,y)} \xi_{x,y} \right\} \tag{6.9}$$

$$\begin{aligned} \text{Constraints}: \quad & y(\tilde{\theta}^T \tilde{x}) \geq 1 - \xi_{x,y} \\ & \xi_{x,y} \geq 0 \end{aligned} \tag{6.10}$$

and for all (x, y). These slack variables $\xi_{x,y}$ allow constraints to be violated **but** for a cost/penalty. Additionally, the new λ coefficient/parameters attached to the first term acts as a regularizer which balances how much error we would allow past our margin. The **dual problem** is presented

as follows:

$$\text{Objective Function}: \quad \max_{\alpha} \left\{ \sum_{(x,y)} \alpha_{x,y} - \frac{1}{2} \sum_{(x,y)} \sum_{(x',y')} \alpha_{x,y} \alpha_{x,'y'} y y' (\tilde{x}^T \tilde{x}') \right\} \tag{6.11}$$

$$\begin{aligned} \text{Constraints}: \quad & \frac{1}{\lambda} \geq \alpha_{x,y} \geq 0 \\ & \sum_{(x,y)} \alpha_{x,y} = 0 \end{aligned} \tag{6.12}$$

Chapter 7

Deep Learning

Deep learning is currently a state of the art technology and is widely used in both research and applied areas. So what is deep learning? It is a form of machine learning used to handle complex non-linear problems. It was developed based on our understanding of the human brain and is a bio-inspired multilayer neural network. This neural network has an input layer, which will then be fed through the network (hidden processing layers which consist of 'neurons') and then finally produce an output at the output layer. A visualization of this is given in the figure below.

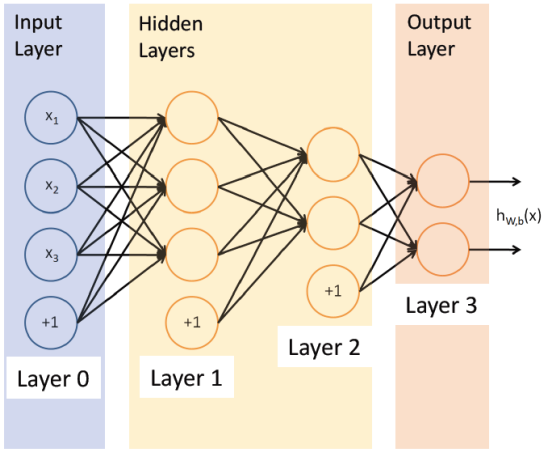

Figure 7.1: Neural Network Visualization

Examples of deep learning applications would be facial recognition, handwriting recognition, etc. Some useful additional online materials can be found here.

§7.1 FeedForward Networks

We begin our study of deep learning by looking at some essential mathematics required for a formal understanding of the subject. First we define the *neuron*.

Definition 7.1.1. Neuron: *A neuron is an activation function f (generally non-linear), which takes an input weighted sum of a feature vector and outputs the function value given this weighted sum input.*

$$h_{w,b}(x) = f(w^T x) = f(\sum_{i=1}^{d} w_i x_i + b) \tag{7.1}$$

*Depending on function f, neurons can be either be real-valued or binary-valued (probabilistic or deterministic) but will **never** produce a negative value.*

It is common that we use the *ReLu* (rectified linear unit $f(z) = \max\{z, 0\}$) as our activation function as is has been shown to produce good results experimentally.

§7.1.1 Multi-Layered Neural Network

Each neuron can be represented by a node in the graph, for which the edges have weights attached to them. So we have $w_{ji}^{(l)}$ as the weight from the i-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer. Then the associated output of the j-th neuron in the l-th layer is given by:

$$\boxed{a_j^{(l)} = f(\sum_{i \in \text{inputs}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)})} \tag{7.2}$$

For the forward propagation of data processing through the neural network, we have:

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)} \tag{7.3}$$

$$a^{(l)} = f(z^{(l)}) \tag{7.4}$$

where $z^{(l)}$ is the (already sorted) *weighted input* to the neurons in layer l and $a^{(l)}$ is the activation of the l layer neurons. Additionally, we call the arrangement of neurons (e.g. number of neurons in each layer) the *neural network architecture*.

§7.2 Backpropagation

The premise of backpropagation is that during the training phase, after we do the forward pass/propagation, we want to compute some loss function at the last layer and feed this information back to the earlier layers so we can amend the weights at each neuron such that the training loss is minimized. As such, we define the training loss as follows:

Definition 7.2.1. Backpropagation Point Loss:

$$J(W,b;x,y) = \frac{1}{2}\|h_{w,b}(x) - y\|^2 + \frac{\lambda}{2}\sum_{l=1}^{L}\sum_{i=1}^{S_l-1}\sum_{j=1}^{S_l}(w_{ji}^{(l)})^2 \tag{7.5}$$

where the regularization term here suppresses overly large weights (a sum over all the squared-weights in the neural network). λ is known as the weight decay regularizer.

Definition 7.2.2. Backpropagation Training Loss:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}\left\|h_{w,b}(x^{(i)}) - y^{(i)}\right\|^2) + \frac{\lambda}{2}\sum_{l=1}^{L}\sum_{i=1}^{S_l-1}\sum_{j=1}^{S_l}(w_{ji}^{(l)})^2 \tag{7.6}$$

The core of backpropagation is to understand how changing the weights and biases in a network changes the cost function. To do this, we require computing the partial derivative $\nabla_w J$ (or $\nabla_b J$) of the cost function J with respect to any weight w (or bias b) in the network. It is also useful to define an intermediate quantity $\delta_j^{(l)}$ which measures the error of in the j-th neuron in the l-th layer.

We can think of the process as having a little demon at the j-th neuron in the l-th layer 'perturbing' the inputs. As inputs enter the neurons, the demon does a change by $\triangle z_j^{(l)}$ to the neurons weight inputs so that the output becomes $f(z_j^{(l)} + \triangle z_j^{(l)})$ instead of just $f(z_j^{(l)})$. This change propagates through later layers in the network, finally causing the overall loss to change by an amount $\frac{\partial J}{\partial z_j^{(l)}} \triangle z_j^{(l)}$. So the affect of this demon on the cost function is governed by the size of the value $\left|\frac{\partial J}{\partial z_j^{(l)}}\right|$.

Motivated by this story, we define the error $\delta_j^{(l)}$ as follows:

$$\boxed{\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}}} \tag{7.7}$$

and $\delta^{(l)}$ is used to denote the vector of errors associated with layer l. As such, backpropagation will allow us to compute this error quantity for every layer and then relate that to the quantities of interest $\nabla_w J$ and $\nabla_b J$. Conventionally, we say that a neural network has L layers, so computing the error in the output layer would mean we are computing $\delta_j^{(L)}$.

$$\begin{aligned}
\delta_j^{(L)} &= \frac{\partial J}{\partial z_j^{(L)}} \\
&= \frac{\partial J}{\partial a_j^{(L)}}\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \\
&= \frac{\partial J}{\partial a_j^{(L)}} f'(z^{(L)})
\end{aligned}$$

To restate this for clarity, we have:

$$\boxed{\delta_j^{(L)} = \frac{\partial J}{\partial a_j^{(L)}} f'(z^{(L)})} \tag{7.8}$$

Since $a^{(L)} = f(z^{(L)})$. The first term on the right measures how fast the cost is changing as a function of the j-th output activation. The second term on the right measures how fast the activation function f is changing with $z_j^{(L)}$. Recalling that $J = \frac{1}{2}\sum_j (y_j - a_j^{(L)})^2$, so we have that $\frac{\partial J}{\partial a_j^{(L)}} = (a_j^{(L)} - y_j)$ and $\delta_j^{(L)} = (a_j^{(L)} - y_j) * f'(z^{(L)})$ where $*$ denotes elementwise multiplication. Now we want to look at the relation of the error in an l-th layer with the error in the next $(l+1)$-th layer. This is relation is given by:

$$\boxed{\delta^{(l)} = \left((W^{(l+1)})^T \delta^{(l+1)}\right) \cdot f'(z^{(l)})} \tag{7.9}$$

where W is the neuron weight matrix. So starting with computing the error at the output layer L and utilizing the relation above, we can recursively compute the errors in every layer of the neural network. From this, how do we tweak the parameters (w and b) appropriately so as to optimize our neural network? First we start by presenting the following propositions:

1. $$\boxed{\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l)}} \tag{7.10}$$

   The proof for this is rather simple by employing the chain rule and recalling that $z_j^{(l)} = \sum_i (w_{ji}^{(l)} a_i^{(l-1)}) + b_j^{(l)}$.

2. $$\boxed{\frac{\partial J}{\partial w_{ji}^{(l)}} = a_i^{(l-1)} \delta_j^{(l)}} \tag{7.11}$$

   Similarly, the proof for this is also rather simple by just recalling the relevant definitions.

With these, we can formalize the full backpropagation algorithm.

Backpropagation Algorithm:

1. **Input x:** We initialize the weights of each neuron and the activation $a^{(0)}$ of the first layer.
2. **Feedforward:** For each layer $l \in [1, L]$, we compute $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$.
3. **Output Error:** We then compute the error in the output layer $\delta_j^{(L)} = \frac{\partial J}{\partial a_j^{(L)}} f'(z^{(L)})$.
4. **Backpropagate the Error:** For l going from L to 1, we then compute the errors for each layer $\delta^{(l)} = \left((W^{(l+1)})^T \delta^{(l+1)}\right) \cdot f'(z^{(l)})$.
5. **Output:** We compute the gradient of the cost functions $\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l)}$ and $\frac{\partial J}{\partial w_{ji}^{(l)}} = a_i^{(l-1)} \delta_j^{(l)}$.

With the output of the backpropagation algorithm, we can apply the SGD algorithm in order to find the optimal parameters which minimize the training loss J.

An example of a neural network implementation for deep learning is an *autoencoder*. The idea of an autoencoder is that we are trying to train a multilayer neural network to reconstruct the input from a dimension reduced *representation* (unsupervised learning). Some strategies for dimensionality reduction are *few hidden neurons* and *sparse activations*. This will be touched on in the next chapter.

Note: Although the backpropagation technique will always allow for a convergence to a solution, the solution may not be the global optimal solution unless the error space is strictly convex.

Chapter 8

Generative Models

Generative models are a method of learning a data distribution in an unsupervised fashion. The goal is to predict the true data distribution of the training data set so that we can generate new data points which follow this distribution. Two of the most common approaches are the ***Variational Autoencoders (VAE)*** *and* ***Generative Adversarial Networks (GAN)****. VAE works by attempting to maximize the lower bound of the data log-likelihood whereas GAN finds an equilibrium between generator and discriminator modules. These will be touched on in greater detail through this chapter.*

§8.1 Some Essential Math

The most commonly found distribution would be the *Gaussian* or *normal* distribution. In machine learning, it is not often that our features have just a single dimension. As such, we look at the Gaussian for general d-dimensional multivariate problems.

Definition 8.1.1. Multivarate Gaussian: *The multivariate Gaussian for a vector of variables/states $x \in \mathbb{R}^d$, given the parameters $\mu \in \mathbb{R}^d$ and $\Sigma \in \mathbb{R}^{d\times d}$ (positive definite) is a probability density function defined as:*

$$p(x|\mu,\Sigma) = \frac{1}{(2\pi)^{d/2}} \exp\Big(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\Big) \tag{8.1}$$

where μ is the mean vector of the observed data and Σ is the ***covariance matrix****.*

Definition 8.1.2. Covariance Matrix: *Given a vector X whose row entries are random variables X_j, the covariance matrix is defined as:*

$$\Sigma_{ij} = E[(X_i-\mu_i)(X_j-\mu_j)] \tag{8.2}$$

So the ij-th entry is the covariance (joint variability) of 2 random variables X_i and X_j. The inverse covariance matrix Σ^{-1} is known as the ***concentration*** *or* ***precision*** *matrix.*

Computing the Covariance Matrix:

In machine learning, we are given data sets in which for a given random variable X_j, we have an n sized set of observed values x_j of this random variable. As such, our vector of random variables becomes a $(n \times d)$ matrix whose j-th column is the vector of observations of random variable X_j.

As such, we compute our covariance matrix as follows:

1. Compute the mean of each random variable $\mu_j = \frac{1}{n}\sum_{i=1}^{n}(X_j)_i$ to give us a vector of averages $\mu = \{\mu_1, \mu_2, ..., \mu_d\}^T$.
2. Convert μ in a matrix $\bar{\mu}$ via the following operation $\bar{\mu} = \mu * \text{row}(1; d)$, where $\text{row}(1; d)$ is a d-length row vector of ones.
3. Finally we compute the covariance matrix via:

$$\boxed{\Sigma = \frac{1}{2}(X - \bar{\mu})^T(X - \bar{\mu})} \tag{8.3}$$

The most ideal form of a Gaussian is known as a *spherical Gaussian*, since its level curves map out hyperspheres when projected on its $d - 1$ domain.

Definition 8.1.3. Spherical Gaussian:

$$p(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{1}{2\sigma^2}\|x - \mu\|^2\right) \tag{8.4}$$

Essentially, it is a standard Gaussian but with the covariance matrix being diagonal with equal variances:

$$\Sigma = \begin{bmatrix} \sigma^2 & 0 & \dots & 0 \\ 0 & \sigma^2 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \sigma^2 \end{bmatrix} \tag{8.5}$$

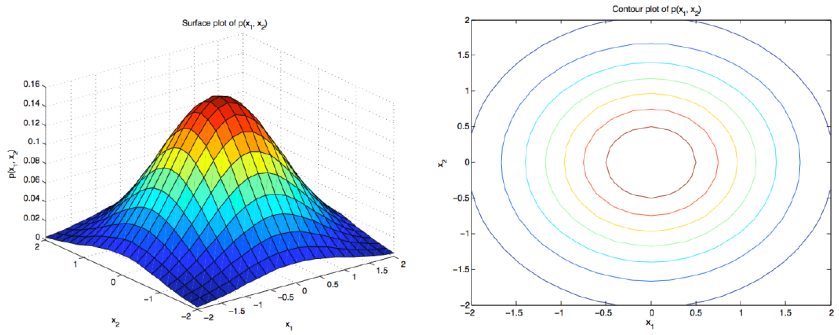

Figure 8.1: Spherical Gaussian of 2 Variables

If the covariance matrix is diagonal but with non-constant variances along its main diagonal,

it would stretch/compress the hypershere along its the parameter $(x_1, x_2, ...)$ axes. A non-diagonal covariance matrix would result in hyperspheres that we stretch/compressed along arbitrary axes.

§8.2 Maximum Likelihood Estimates (MLE)

For our generator model, we will be utilizing the concept of *maximum likelihood estimation* which is a means of estimating the parameters of a statistical distribution model. This is done in a model for which we have the parameters $\theta \in \mathbb{R}^d$ with associated distributions $\mathbb{P}(x|\theta)$. We are also given the set of observations $S = \{x^{(1)}, x^{(1)}, ..., x^{(n)}\}$ for which these are used to approximate the parameters $\hat{\theta} \in \mathbb{R}^d$ that best describes the data in S.

Definition 8.2.1. Likelihood Function: *We define the likelihood function, or simply the likelihood, as:*

$$p(S|\theta) = \prod_{x \in S} p(x|\theta) \tag{8.6}$$

Which is the probability density function (distribution) given some fixed parameters θ.

Definition 8.2.2. Maximum Likelihood Estimate: *The maximum likelihood estimates are defined as as:*

$$\hat{\theta} = \text{argmax}_\theta p(S|\theta) \tag{8.7}$$

which are the parameters that maximize your likelihood.

Maximizing Log Likelihood:

An equivalent optimization problem for likelihood maximization is minimizing the negative log of the likelihood.

$$\begin{aligned} \min\{-\frac{1}{n}\log \mathbb{P}(x|\theta)\} &= \min\Big\{-\frac{1}{n}\log \prod_{x \in S} p(x|\theta)\Big\} \\ &= \min\Big\{\frac{1}{n}\sum_{x \in S}\frac{1}{\log p(x|\theta)}\Big\} \end{aligned} \tag{8.8}$$

It is then convenient to define this objective function as the *training loss*.

Definition 8.2.3. MLE Point Loss:

$$\mathcal{L}(\theta; x) = -\log \mathbb{P}(x|\theta) \tag{8.9}$$

Definition 8.2.4. MLE Training Loss:

$$\mathcal{L}_n(\theta; S) = \frac{1}{n}\sum_{x \in S}\frac{1}{\log \mathbb{P}(x|\theta)} \tag{8.10}$$

Let us now look at an example of an application of least likelihood estimation.

Example:

Suppose we have a data set $S = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$ that we assume to be independent and identically distributed with a spherical Gaussian distribution (means $\mu \in \mathbb{R}^d$ and variance σ^2). As previously defined, our loss function is given by:

$$\begin{aligned}
\mathcal{L}_n(\mu, \sigma^2; S) &= -\frac{1}{n}\sum_{x\in S} \log p(x|\mu, \sigma^2) \\
&= -\frac{1}{n}\sum_{x\in S} \log \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{1}{2\sigma^2}\|x-\mu\|^2\right) \\
&= \frac{d}{2n}\sum_{x\in S} \log\left(2\pi\sigma^2\right) + \frac{1}{n}\sum_{x\in S}\frac{\|x-\mu\|^2}{2\sigma^2} \\
&= \frac{d}{2}\log\left(2\pi\sigma^2\right) + \frac{1}{2n\sigma^2}\sum_{x\in S}\|x-\mu\|^2
\end{aligned} \tag{8.11}$$

To get the minimum likelihood estimator, we apply the necessary conditions and solve for the parameters $\hat{\mu}$ and $\hat{\sigma^2}$. First looking at the parameter μ, we have:

$$\begin{aligned}
&\nabla_\mu \mathcal{L}_n = 0 \\
\Rightarrow\ &\frac{\partial \mathcal{L}_n}{\partial \mu_j} = -\frac{1}{n\sigma^2}\sum_{x\in S}(x_j - \mu_j) = 0 \\
\Rightarrow\ &\boxed{\hat{\mu} = \frac{1}{n}\sum_{x\in S} x}
\end{aligned} \tag{8.12}$$

And similarly for σ^2:

$$\begin{aligned}
&\frac{\partial \mathcal{L}_n}{\partial \sigma^2} = 0 \\
\Rightarrow\ &\frac{d}{2\sigma^2} - \frac{1}{2n\sigma^4}\sum_{x\in S}\|x-\mu\|^2 = 0 \\
\Rightarrow\ &\boxed{\hat{\sigma}^2 = \frac{1}{nd}\sum_{x\in S}\|x-\mu\|^2}
\end{aligned} \tag{8.13}$$

§8.3 Variational Autoencoders (VAE)

We will be looking at both supervised and unsupervised learning paradigms, and the goal of a generative model for both these contexts. Recall that in supervised learning, we are trying to map feature vectors x to label vectors y. Whereas for unsupervised learning, we are trying to look for underlying/hidden *structure* within the data sets. Knowing this, we can use unsupervised learning to understand the hidden structures in our visual world!

'What I cannot create, I do not understand.'

– Richard Feynman

In the context of generative models and image generation, we want to have algorithms that generate image distributions such that the $p_{model}(x)$ is similar to $p_{data}(x)$. How do we do this? We can use the system of an *autencoder*. The goal of an autoencoder is to approximate an *identity function*, that is to say the the autoencoder tries to learn a function $h_{W,b}(x) \approx x$. This may seem like a trivial task, but the point is that this function will give us insights to structures within the data.

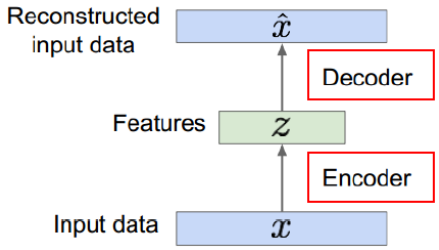

Figure 8.2: Autoencoder Diagrammatic Flow

First, we explore the use of what is called a *sparse autoencoder*. Sparse autoencoders make use of a sparsity constraint on the hidden neurons, for which this imposition still allows us to discover interesting structure in the data, even if the number of hidden units is large. Along with this constraint, we would like the average activation $a_{ji}^{(l)}$ of each hidden neuron to be small (usually 0.05). This low average activation value can be written as:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} a_j(x^{(i)}) \tag{8.14}$$

where a_j denotes the activation value of hidden unit j and m is the number of data samples. We introduce a *sparsity parameter* $\rho = 0.05$ such that $\hat{\rho}_j = \rho$ (i.e. all activations of neuron j are low/close to 0.05). To achieve this, we add an additional term to our objective/loss function that will be 0 when $\hat{\rho}_j = \rho$ and increase monotonically as ρ diverges from $\hat{\rho}_j$. The function we utilize for this is the *Kullback-Leibler-divergence* (KL-divergence).

Definition 8.3.1. Kullback-Leibler Divergence: *KL-divergence is a standard function that gives a measure of the difference between 2 distributions and is defined as:*

$$KL(p||\hat{p}_j) = p \log \frac{p}{\hat{p}_j} + (1-p) \log \frac{1-p}{1-\hat{p}_j} \tag{8.15}$$

As such, our loss function is now given by:

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{i=1}^{s} KL(p||\hat{p}_j) \tag{8.16}$$

where s is the number of neurons, β is another regularizer and $J(W, b)$ is the standard backpropagation loss defined in definition 7.2.

§8.4 Generative Adversarial Networks (GAN)

'GAN is the most important upcoming breakthrough in deep learning.'

– Yan LeCun

The basic architecture of a *GAN* model is first, we input a random noise feature vector into a generative model G which produces an output (e.g. image). This output is then fed into a discriminator D which compares/discriminates the generated output with an real observed data. The output of the discriminator D is simply binary (output $\in \{0, 1\}$), and sorts the outputs 0 if the output is classified as real data, and 1 if it is classified fake.

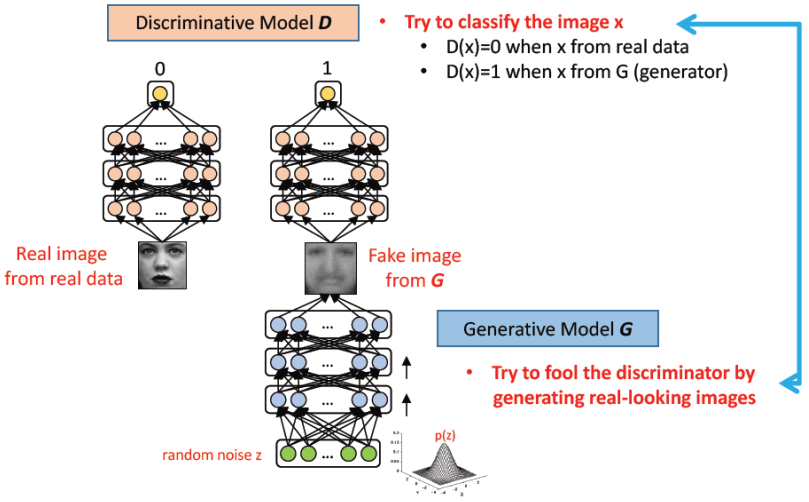

Figure 8.3: GAN Diagrammatic Structure

So the question is, how can we generate fake images that look real from random feature vectors? Ian Goodfellow came up with the idea that we can achieve this via attempting to 'fool' the disciminator into making mistakes. Essentially, we are going to pit the generator and the discriminator against each other in sorts of a competitive game. This is again can be mapped into an optimization problem as follows.

Adversarial Otimization:

let's look at images. As the name suggest, the idea is that now we think of D and G as players competing against each other. D wins by correctly distinguishing real and generated images wheres G wins if it fools D. So what we propose is a turn-based game. D starts by practicing (deep learning) on a set of generated images from G. After sufficient practice (optimizing its neuron weights for discrimination), the turn is then passed to

G which practices fooling D (optimizing its neuron weights for real-looking generated images). This game is then repeated until the generated images look amazingly real. Formally, this process is presented as follows:

First, we define a prior probability distribution $p_z(z)$ on the input noise variables z. The **generator** network then gives a differentiable mapping to the data space $G(x;\theta_g)$ which we can then use to learn our generators distribution p_g.

Next, we define a map for our **discriminator** $D(x;\theta_d)$ which outputs 0 or 1 for 'real' or 'generated' classification (could also be a continuous output $\in [0,1]$ that will give a probability of the images being real or generated).

We then define the **objective function** as:

$$V(G,D) = \mathbb{E}_{x\sim p_{data}(x)}\big[\log D(x;\theta_d)\big] + \mathbb{E}_{z\sim p_z(z)}\big[\log\big(1 - D[G(z;\theta_g)]\big)\big] \tag{8.17}$$

With this, we can run our little competition between the 2 networks.

1. Optimize $V(G,D)$ while fixing G (in favour of D):

$$\max_D\{V_G(D)\} = \max_D\Big\{\frac{1}{m}\sum_{i=1}^{m}\Big[\log D(x;\theta_d) + \log\big(1 - D[G(z;\theta_g)]\big)\Big]\Big\} \tag{8.18}$$

   To do this, we apply the gradient ascent (SGA) using the gradient with respect to parameters θ_d, $\boldsymbol{\nabla}_{\theta_d} V_G(D)$.

2. Optimize $V(G,D)$ while fixing D (in favour of G):

$$\min_G\{V_D(G)\} = \min_G\Big\{\mathbb{E}_{z\sim p_z(z)}\big[\log\big(1 - D[G(z;\theta_g)]\big)\big]\Big\} \tag{8.19}$$

   To do this, we apply the gradient descent (SGD) using the gradient with respect to parameters θ_g, $\boldsymbol{\nabla}_{\theta_g} V_D(G)$.

The algorithm (pseudo-code) can be summarized as follows:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

Update D

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

Update G

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 8.4: Overview of the GAN Algorithm

Chapter 9

Kernel Methods and Convolutional Neural Networks

The motivation of using Kernel methods is that during classification, the 2 classes are not linearly separable. So we project the data from the original feature space into a higher dimensional space such that we can find linear decision boundaries. If we are able to find some projection function that does this effectively, then we are able to separate classes in linear way which we are familiar with.

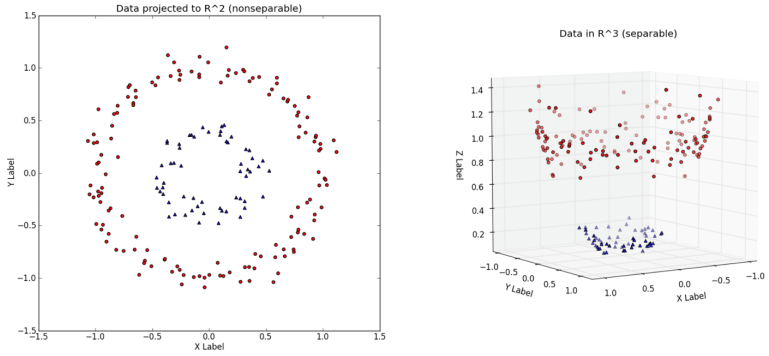

Figure 9.1: Higher Dimensional Projections for Linear Separability

§9.1 Kernel Methods

A basic concept in machine learning is the *dot product*. You often do dot products of the features of a data sample with some weights w, the parameters of your model. Instead of doing explicitly this projection of the data in 3D and then evaluating the dot product, you can find a kernel function that simplifies this job by simply doing the dot product in the projected space for you, without the need to actually compute projections and then the dot product. This allows you to find a complex non-linear boundary that is able to separate non-linearly separable classes in the data set.

§9.1.1 Feature Mapping

In order to fully utilize the power of kernel methods, what we normally do is first construct a map that takes the feature vectors from our original vector space into a higher dimensional space that ensures linear separability. This map is called a *feature map* and is generally denoted as $\phi(x)$,

$$\phi : \mathcal{X} \to \mathcal{V} \tag{9.1}$$

where x is the feature vector from the original vector space $\mathcal{X}$ and $\mathcal{V}$ is the new linearly seperable vector space equipped with an inner product $\langle\ ,\ \rangle_{\mathcal{V}}$. To better understand this, consider the following **example**. Let's say with have a data set of 2D feature vectors $x = (x_1, x_2)$ and supposed the data is **not** linearly separable in the 2D space. We can construct some higher dimensional space such that the data becomes linearly separable with the vector function ϕ and classifier $h(x; \theta, \theta_0)$ defined as:

$$\phi(x) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2) \tag{9.2}$$

$$h(x; \theta, \theta_0) = \text{sign}\big(\theta \cdot \phi(x)\big) = \text{sign}\Big(\sum_{j=1}^{6} \theta_j \phi_j(x)\Big) \tag{9.3}$$

We have now effectively allowed for linear separability. However, the size of our space has largely increased, increasing the complexity of our algorithm (complexity of inner products is $\mathcal{O}(d)$). So we look to *kernel functions* to help us with this dimensionality problem.

Definition 9.1.1. Kernel Function: *A function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a kernel function iff:*

1. *It is symmetric: $K(x, y) = K(y, x)$ for all $x, y \in \mathbb{R}$*
2. *Given $n \in \mathbb{N}$ and $x^{(1)}, x^{(2)}, ..., x^{(n)} \in \mathbb{R}^d$, the **Gram matrix** $\hat{K}$ with entries $\hat{K}_{ij} = K(x^{(i)}, x^{(j)})$ is positive semidefinite (eigenvalues ≥ 0).*

What is essentially happening in a Kernel function is that it is computing the inner product of $\phi(x)$ and $\phi(x')$ on the inner product space $\mathcal{V}$, but without having to explicitly specify the feature map ϕ and thus not having to explicitly compute this higher dimensional inner product! The hard part is defining an appropriate inner product function $\langle \phi(x), \phi(x') \rangle_{\mathcal{V}}$ that achieves this. Some examples of commonly used Kernel functions are given below.

Kernel Examples:

1. **Linear Kernel:** $\boxed{K(x, x') = x \cdot x'}$.
2. **Polynomial Kernel:** $\boxed{K(x, x') = (x \cdot x' + 1)^k}$.
3. **Radial Basis Kernel:** $\boxed{K(x, x') = \exp\Big(-\frac{1}{2}\|x - x'\|^2\Big)}$
   For the radial basis kernel, the higher dimensional space is infinite dimensional but the kernel function allows us reduce that to a finite dimensional space.

Kernel functions have several useful **properties**:

1. Kernels can be constructed manually.
2. Inner (scalar) products $\langle x, x' \rangle$ are kernels.
3. Constant maps $K(x_1, x_2) = 1$ are kernels.
4. The product of kernels is a kernel $K(x, x') = K_1(x, x')K_2(x, x')$.
5. For every function $\phi : X \to \mathbb{R}$ the product $K(x, x') = \phi(x)\phi(x')$ is a kernel.
6. Linear combinations of kernels $K(x, x') = a_1K_1(x, x') + a_2K_2(x, x')$ with **positive** coefficients are kernels.

As earlier mentioned but not discussed, the *kernel trick* is a strategy that allows us to only have to compute the kernel function (not the actual feature maps $\phi(x)$) to perform a learning algorithm. This is useful because in general, computing the kernel is much less computationally heavy than computing the feature map.

§9.2 Convolutional Neural Networks

Convolutional neural networks act as image classifiers and does so by learning and identifying features of an image. To do this, we first note that often times, there are patterns which are localized and do not require processing the entire image to pick out. How then do we use this to our advantage for machine learning implementations? A *convolutional neural network* takes advantage of this beautifully, and we will work through learning how it does so.

§9.2.1 Convolutional Filters and Layers

A CNN is a neural network with some *convolutional layers* (and possibly some other layers). Each of these convolutional layers have a number of *convolutional filters* that do the *convolutional operations*. We can treat these convolutional filters as the weights that we want to optimize using back propagation (just as in a standard deep neural network). A convolution is essentially the same operation as a dot/inner product, so each filter (acting as a neuron) maps features of an image to a scalar value (which is essentially the neuron activation value to be fed into the next layer). Before we move on, it is important that we familiarize ourselves with some common terminologies used by people in the CNN field:

- **Filter**: A pixel grid (with weights attached to each pixel) smaller than the whole image that we use to compute it's dot product with the image pixels it is 'placed over'.
- **Stride**: Number of pixels the filter shifts by for each convolution operation.

Ideally, the filter detects some pattern present in the image and weights that pattern positively. All other entries in the filter matrix is weighted down (negative). As such, the output of the filter's inner product gives the likelihood of the feature being at a specific location (more positive implies higher likelihood). It is best to use an example to illustrate this.

Example

Let's say that we want to look for a particular **local** feature in an 6×6 pixel image that

corresponds to the following pattern:

$$\begin{bmatrix} * & & \\ & * & \\ & & * \end{bmatrix} \tag{9.4}$$

For simplicity, we also consider an image that only has binary (0 or 1) image pixel values. What we could is use a 3×3 pixel filter that has weights which 'look out' for this pattern, and move this filter around to generate a *feature map*. Our filter could be constructed as follows:

$$\begin{bmatrix} +1 & -1 & -1 \\ -1 & +1 & -1 \\ -1 & -1 & +1 \end{bmatrix} \tag{9.5}$$

where we see that the diagonal pattern we are looking for would receive positive weightings if our filter is applied to it, and all other unwanted features are down weighted. So let's say our image is given as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{9.6}$$

We can move our filter along this image pixel matrix with a stride of 1 starting from the top left most corner. Doing this will allow us to generate a 4×4 feature map who's entries are simply the inner product values of the filter applied to the corresponding portion of the image. The feature map generated for this is given as follow:

$$\begin{bmatrix} 3 & -1 & -3 & -1 \\ -3 & 1 & 0 & -3 \\ -3 & -3 & 0 & 1 \\ 3 & -2 & -2 & -1 \end{bmatrix} \tag{9.7}$$

With these feature map, we simply extract the entries with the highest output values and infer that these locations on the image are most likely to contain the feature we are looking for.

§9.2.2 Max Pooling

In general, images can be very large (many pixels) for which not all the information from every pixel is actually required to pick out essential features in the image. In order to reduce the computation required to process large images, we utilize a technique known as *max pooling*. Essentially, max pooling works by taking the maximum value of smaller windows over the feature maps. Let's illustrate this with an example.

Example:

Consider the feature map generated from our previous example (equation 9.7). Now, we can perform max pooling by using some 2×2 window and a stride of 1, which will allow us to generate the following 'compressed' feature map:

$$\begin{bmatrix} 3 & 1 & 0 \\ 1 & 1 & 1 \\ 3 & 0 & 1 \end{bmatrix} \tag{9.8}$$

What we essentially did here is similar to what we did for feature mapping, but instead of taking the inner product of some filter on the applied site, we simply extract the highest data value in the window. As such, the compressed feature map above is the output of max pooling.

So we see that a CNN compresses a fully connected network in 2 ways:

1. Reduce the number of connections and shared weights on the edges (in the convolutional layer).
2. Reduces the complexity of the data (via max pooling which reduces the size of the feature map).

Chapter 10

Recurrent Neural Networks

Recall that in a convolutional neural networks (CNN), inputs and outputs are independent of each other and no memory of the previous input is stored. However in recurrent neural networks (RNN), it has 'memory' of the inputs and thus allows for learning ***sequential data****. As such, we term RNNs a* ***sequence model****. With this, it can be applied to tasks like speech recognition, machine translation and time-series prediction. As a side note, RNNs are* ***not*** *the only sequence model to solve the sequential data problem. Examples of other sequence models are the auto-regressive model, feed-forward neural nets, hidden Markov model, etc. RNNs are also useful for non-sequential input data and not limited to just the sequential paradigm.*

§10.1 Vanilla RNN Unit/Cell

There are several things that make an RNN so effective as a sequence model:

- **Distributed hidden states** that allow RNNs to store a lot of information about the past efficiently.
- RNNs have **non-linear dynamics** that allows them to update their hidden state in complex ways.
- There is no need to infer the hidden state, they are purely **deterministic**.
- RNNs utilize **weight sharing**.

In general, an RNN consists of a recurrent core cell that takes some input x, feeds this input into the RNN, which has some internal hidden state, and that internal hidden will be updated every time the RNN reads a new input. It is common that we will also want our RNN to produce an output at every iteration/time step. The recurrence relation of this hidden state is given the functional form:

$$h_t = f_W(h_{t-1}, x_t) \tag{10.1}$$

Conceptually, we can think of RNNs in 2 ways:

1. We can think of it having this hidden state that feeds back on itself recurrently (conventional presentation but may be a little confusing).

2. Alternative, we can consider an 'unrolled' version of this architecture over multiple time steps. At the first time step we have some initial hidden state h_0 and some input x_1. These will go into our f_w function which will output the next hidden state h_1. Then this will be fed in along with x_2. Then this process keeps running like a film roll revealing itself.

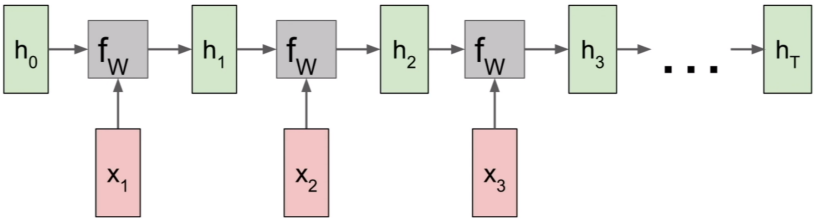

Figure 10.1: Unrolled RNN Representation

Furthermore, we can think of each step having an output y_t and a loss $\mathcal{L}_t$ and the total loss is just the sum of the losses at each time step.

Note: we use the same function f_w and these same weights w at every time step of this computation.

The simplest functional form that one can imagine is called a *vanilla recurrent neural network* (a basic 'flavour'). The hidden state in a vanilla RNN cell takes the form:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{10.2}$$

where W_{hh} and W_{xh} are some weight matrices and we squash this weighted sum into a tanh to create some non-linearity. We can also just concatenate the W_{hh} and W_{xh} matrices into a larger W matrix to give:

$$\boxed{h_t = \tanh\left(W\begin{bmatrix}x_t\\h_{t-1}\end{bmatrix}\right), \quad W = [W_{xh}\, W_{hh}]} \tag{10.3}$$

§10.1.1 Vanilla RNN Forward Pass

The vanilla RNN consists of a forward pass and backward pass just as per neural networks which utilize *backpropogation*. We will first be presenting the steps required in the forward pass step. As seen in figure 10.2 where the network flows from left to right, we have the unrolled sequence of inputs $x_1, x_2, ...$ being fed into each vanilla cell along with the previous hidden states $h_0, h_1, ...$ Furthermore, the jth vanilla cells outputs $y_t = F(h_t)$ which is computed solely based on the current hidden state h_t for which a point loss $C_t = \text{Loss}(y_t, GT_t)$ is computed that will later be used in the backward pass.

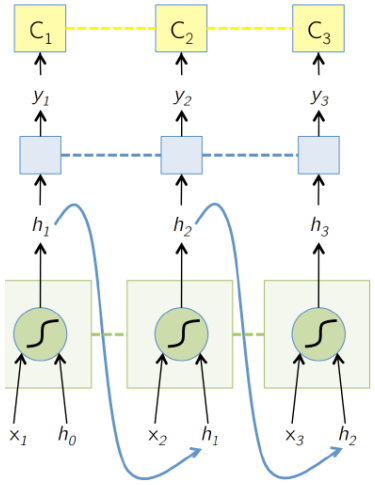

Figure 10.2: Vanilla RNN Forward Pass Visualization

Let us now look at a common paradigm in which one might want to employ a recurrent neural network, *sentiment classification*.

§10.1.2 Sentiment Classification

Supposed we want to classify a restaurant review from *Yelp!* or a movie review on *IMDB*. For simplicity, we will be looking at a classification into binary outputs representing positive or negative sentiment. To do this, we can implement the following architectures:

Sentiment Classification Architecture 1:

1. We feed in each words of the sentence into individual vanilla cells (each word is an input x_t).
2. Then the hidden state outputs are fed (in sequence) to the adjacent RNN cell.
3. At the end of the sentence (RNN), only the last RNN cell's hidden state output is processed by a linear classifier to interpret the review/sentiment.

A visualization of this is given in the figure (10.3) below:

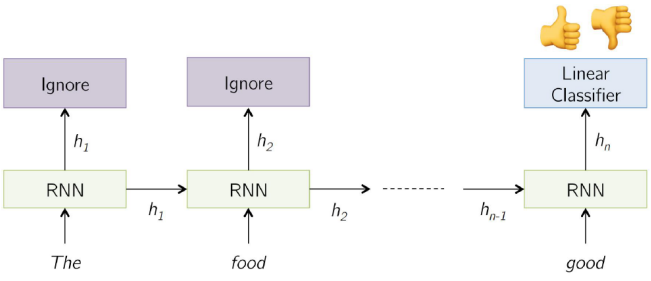

Figure 10.3: RNN Sentiment Classifier Architecture

Alternatively, we could also adopt an architecture that incorporates the hidden states at every time step (RNN cell). This architecture works as follows:

Sentiment Classification Architecture 2:

1. We feed in each words of the sentence into individual vanilla cells (each word is an input x_t).
2. Then the hidden state outputs are fed (in sequence) to the adjacent RNN cell and also added to the previous hidden state ($h_t + h_{t-1} + ... + h_1$).
3. At the end of the sentence (RNN), we take the sum of all hidden states to be processed by a linear classifier to interpret the review/sentiment.

A visualization of this is given in the figure (10.4) below:

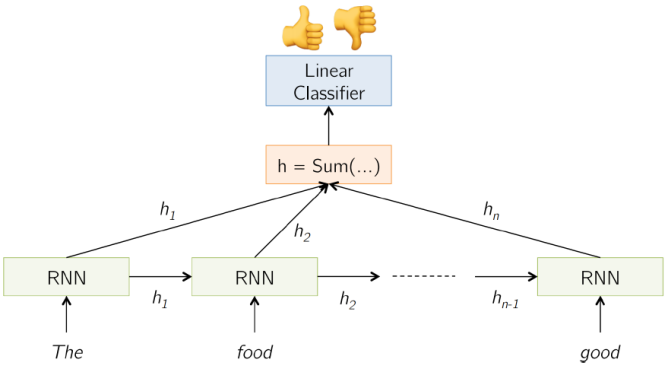

Figure 10.4: Alternate RNN Sentiment Classifier Architecture

§10.1.3 BackPropagation Through Time (BPTT)

Having used the forward pass to generate losses, we now require a backpropagation of this information through our network to update the weights W. Broadly speaking backpropagation refers to two things:

1. The mathematical method used to calculate derivatives and an application of the **chain rule**.
2. The training algorithm for **updating network weights** to **minimize error**.

For our application of this to RNNs, first recall the standard backpropagation procedure as discussed in section 7.2. For an RNN, we backpropagate the error in a similar way but now instead of backpropagating it through the many network layers, we backpropagate it through time steps (so we treat each tie step as another layer). Let us work through doing this.

After the forward pass, we are given the cost function C and the outputs y from which we can compute $\frac{\partial C}{\partial y}$. However to update the weights, we require to know how C changes with W (i.e. $\frac{\partial C}{\partial W}$). As per standard backpropagation, this can be done recursively so we can get the errors at every time step. However, we may run into a problem because we additionally have previous

hidden states as inputs. So at any of the the t time steps, we have:

$$\frac{\partial C_t}{\partial h_1} = \frac{\partial C_t}{\partial y_t}\frac{\partial y_t}{\partial h_1} = \frac{\partial C_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}...\frac{\partial h_2}{\partial h_1} \tag{10.4}$$

This long chain of product terms could prove detrimental to our algorithm because if the functional form of the hidden states is such that $\frac{\partial h_t}{\partial h_{t-1}} > 1$ (or < 1), $\frac{\partial C_t}{\partial h_1}$ could explode (or decay) rapidly!

To fix this, we enforce an *identity relationship* between the hidden states. The relationship is defined as follows:

$$\boxed{h_t = f_W(h_{t-1}, x_t) = h_{t-1} + F(x_t)} \tag{10.5}$$

The above relation ensures that $\frac{\partial h_t}{\partial h_{t-1}} = 1$, so that the error is allowed to 'stably' propagate all the way back through all time step layers. We call this stable propagation *constant error flow*. As a small overview of the combined forward pass and BPTT procedures, the general algorithm looks as follows:

RNN BPTT Overview:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Adjust the weights to minimize the error.
5. Repeat.

§10.2 Long Short-Term Memory (LSTM)

LSTM makes use of the previously mentioned idea of constant error flow for RNNs to create a *constant error carousel* (CEC) which ensures that gradients don't explode/decay. The key component of the CEC is a *memory cell* that acts like an accumulator over time. In this paradigm, instead of computing new hidden states as a matrix product of weights with the old hidden states and inputs, the CEC of an LSTM computes the **difference** between the hidden states. As a result, the gradients are more "well-behaved".

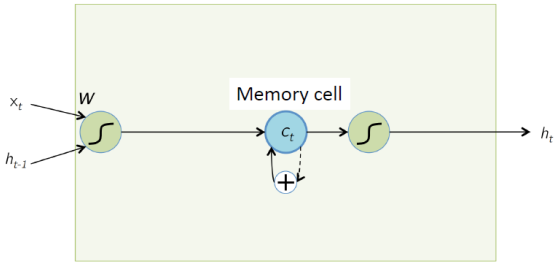

Figure 10.5: Rudimentary Constant Error Carousel

As seen from figure 10.5 above, we have done a little modification to our vanilla RNN cell by adding a *memory cell* and sandwiching that between 2 non-linear units. We represent the state of the memory cell as c_t and call it the *cell state* (the cell state with a lowercase c and **not** to be confused with the cost function C). In this model, we define the cell state with a recurrence relation as:

$$c_t = c_{t-1} + \tanh\left(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right) \tag{10.6}$$

The t-th hidden state is then taken to be:

$$h_t = f_W\big(c_t(h_{t-1}, x_t; c_{t-1})\big) = \tanh c_t \tag{10.7}$$

In the original LSTM CEC unit, there is also usually 2 additional gates namely the *input* and *output gates*.

> **Note:** It is important to note that these names come from their position within the CEC but do not imply that they actually input or output data.

Both the input and output gates also take in x_t and h_{t-1} inputs, for which separate weights (W_i and W_o) are applied and is fed through a sigmoid function. The cell state would now be computed as follows:

$$\begin{aligned} &\boxed{c_t = c_{t-1} + i_t * \tanh\left(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right)}, \\ &i_t(h_{t-1}, x_t) = \text{sigmoid}\Big(W_i \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b_i\Big) \end{aligned} \tag{10.8}$$

where we recall that $*$ stands for element wise multiplication. The current hidden state h_t is then computed as follows:

$$\begin{aligned} &\boxed{h_t = o_t * \tanh c_t}, \\ &o_t(h_{t-1}, x_t) = \text{sigmoid}\Big(W_i \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b_i\Big) \end{aligned} \tag{10.9}$$

We can think of i_t and o_t simply as x_t and h_{t-1} dependent weights ($\in [0, 1]$) that are added to the non-linear tanh terms. Figure 10.6 below gives a visualization of this LSTM CEC and hopefully a clearer picture of what is going on here.

> **Note:** Element wise multiplication is sometimes denoted with a $\otimes$ instead of $*$ depending on the author, but we will avoid $\otimes$ in these notes since it can be confused with tensor products.

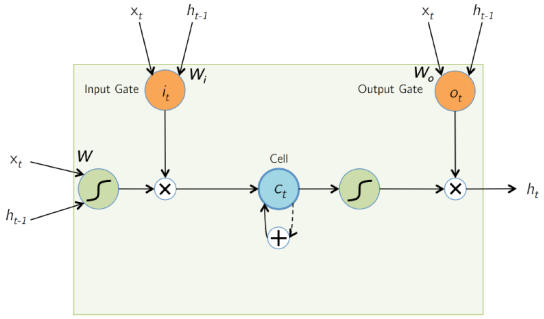

Figure 10.6: Original Constant Error Carousel Architecture

one-hot encoding is when we have a vector of zeros and only one entry being set as 1.

Chapter 11

Expectation Maximization

The premise is as follows. Presume that we are presented with some unlabelled data that comes from a multivariate distributions (several data clusters of some distribution). Our task is to come up with a hypothesis for the parameters of each distribution. The EM algorithm is most commonly adopted by the ***Gaussian mixture model*** *which is similar k-means clustering. One thing to keep in mind for EM is that it is easy to overfit your model because just like in k-means, there is the issue where increasing your number of clusters will always reduce the loss. Hence, it is important to find ways to approximate the number of clsuters.*

§11.1 Generative Gaussian Mixture Model

First, we recall that for k-means, the general overview its algorithm is as follows:

Simplified k-Means Algorithm:

1. Given hard labels, we compute the centroids.
2. Given centroids, we compute the hard labels.
3. Repeat.

Expectation maximization works in a very similar way just that instead of hard labels, we are instead computing *soft labels* (data points could belong to more than one cluster).

Note: For EM Gaussian mixture models, it is convention that we term the clusters as *Gaussians*, so when we say Gaussian, we are referring to the data clusters.

The process flow starts by first using an algorithm like k-means to pick out Gaussian centers, after which we can assign a set of labels y_j for $j \in \{1, 2, ..., k\}$ each with an associated probability p_j to the data (these probabilities are initialized naively with the *prior distributions*).Prior distributions are found by acknowledging that different Gaussians have different number of data points, hence the Gaussians with more data points have a larger probability to be sampled from. So in the initialization step, p_j is simply related to the number of points.

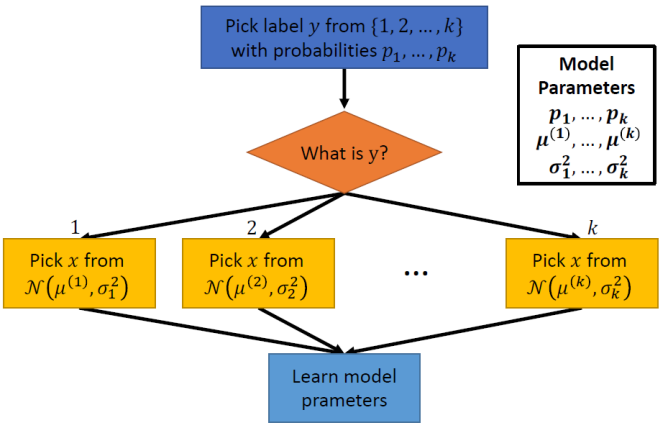

Figure 11.1: Gaussian Mixture Model Process Flow

§11.1.1 Mixture Model and Hidden Labels

The data points are taken to be from Gaussian distributions and hence, are written as $x \sim \mathcal{N}(\mu^{(y)}, \sigma_y^2)$ which produces the observed data set $S_n = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$. The parameters for our model are then given by $\theta = \{p_j, \mu^{(j)}, \sigma_j^2\}$ for $j \in [1, k]$. We then create a Gaussian mixture model by simply taking our data points as coming from the sum of the individual Gaussian distributions. So our task is to predict these individual Gaussians. Utilizing our prior probability distribution, our model pdf is given by:

$$\boxed{p(x|\theta) = \sum_{y=1}^{k} p_y \cdot p(x|y, \theta)} \tag{11.1}$$

$$\text{where} \quad p(x|y, \theta) = \frac{1}{\sqrt{(2\pi\sigma_y^2)^d}} \exp\Big(-\frac{1}{2\sigma_y^2} \Big\| x - \mu^{(y)} \Big\|^2 \Big) \tag{11.2}$$

As such, our objective function (log likelihood) is given by:

$$\mathcal{L}_n(\theta) = \sum_{(x,y) \in S_n} \log\Big(\sum_{y=1}^{k} p_y \cdot p(x|y, \theta) \Big) \tag{11.3}$$

where we want to **maximize** this function. But the difficulty arises from filling in the missing labels $y^{(j)}$ for the $j = 1, 2, ..., k$. We do this by taking the expectation from the current model parameter μ, and then run a learning algorithm that gives a better/updated model parameter μ' (this is like the Gaussian version of k-means). As such, we construct the numerical algorithm as follows:

EM Algorithm:

1. Initialize parameters $\theta = \{p_j, \mu^{(j)}, \sigma_j^2\}$ for $j = 1, 2, ..., k$.

2. Repeat the following steps until convergence:
   (a) *E-Step:* Given parameters θ, compute soft labels $p(y|x)$.
   (b) *M-Step:* Given soft labels $p(y|x)$, compute parametersθ.

In detail, the parameters are initialized as follows:
- $p_y = 1/k$ which is a uniform distribution.
- $\mu^{(y)}$ as the centroids from the k-means algorithm.
- $\sigma_y^2 = \sigma^2$, which is the same sample variance for all y.

The soft labels are computed via Baye's theorem:

$$p(y|x) = \frac{p(y,x)}{p(x)} = \frac{p_y \cdot p(x|\mu^{(y)}, \sigma_y^2)}{\sum_{z=1}^{k} p_z \cdot p(x|\mu^{(z)}, \sigma_z^2)} \tag{11.4}$$

where all the $p(x|\mu^{(j)}, \sigma_j^2)$ is the spherical Gaussian distribution.

As for the log likelihood maximization step, we compute:
- The *effective* number of points with label y, $\hat{n}_y = \sum_{x \in S_n} p(y|x)$ (does not have to be an integer).
- The *effective* fraction of points with label y, $\hat{p}_y = \hat{n}_y / n$.
- The weighted mean of points with the label y, $\hat{\mu}^{(y)} = \frac{1}{\hat{n}_y} \sum_{x \in S_n} x \cdot p(y|x)$.
- The weighted variance of points with label y, $\hat{\sigma}_y^2 = \frac{1}{d\hat{n}_y} \sum_{x \in S_n} p(y|x) \left\| x - \hat{\mu}^{(y)} \right\|^2$

Like k-means, EM clustering may get stuck in local minima. However unlike k-means, the local minima are more favorable because soft labels allow points to move between clusters slowly. An important question to ask now is, how do we choose k (the number of Gaussians)? We can use a validation set.

§11.1.2 Cross-Validation

There are several methods we can use for *cross-validating* that we have not overfit our model. These are presented below:

1. **m-fold cross validation** is when we segment our data set into m sub-data sets. We then use one of those subsets for testing and the remaining $m-1$ subsets for training. We then swap out one of the training subsets for the testing set (permutate the subsets) and repeat this m times.

2. **The marginal likelihood** method introduces the notion of a *Bayesian information criterion* (BIC) which is a new objective function. In this new objective function, we add another term that is dependent on the number of free parameters.

   $$BIC(\theta) = \mathcal{L}_n(\theta) - \frac{k(d+2)-1}{2} \log n \tag{11.5}$$

   where $k(d+2)-1$ is the number of free parameters for Gaussian mixtures. As such, this places a penalty for increasing the number of parameters used in our model.

§ **Midterm Summary** §

In machine learning, we are doing nothing more than learning functions. As a quick recap, we have thus far seen 2 broad categories of machine learning models. These are namely supervised and unsupervised learning. These differ in their presence of labels for the given data set.

So far for **supervised learning**, we have seen classification and regression.

1. Classification:
   Classification aims to learn a classifier functions which is a mapping as follows:
   $$f : \mathbb{R}^d \to \{1, 2, ..., k\} \tag{11.6}$$
   Some examples are the perceptron, neural networks, support vector machines, logistic regression, etc.
2. Regression:
   Regression aims to learn a
   $$f : \mathbb{R}^d \to \mathbb{R}^m \tag{11.7}$$
   Some examples of this are linear regression, ridge regression.

As for **unsupervised learning**, we have seen clustering and

1. Clustering:
   $$f : \mathbb{R}^d \to \{1, 2, ..., k\} \tag{11.8}$$
   An example of this is k-means clusterning.
2. Dimension Reduction:
   $$f : \mathbb{R}^d \to \mathbb{R}^m \tag{11.9}$$
   Some examples are autoencoders (AE), matrix factorization, principle component analysis, etc.

We also looked at generative models. In a generative model, we generate data points $(x^{(j)}, y^{(j)})$ by first generating the $y^{(j)}$ from its distribution, and then generating the corresponding feature vector from that associated distribution. This is written generally as:

$$\mathbb{P}(x^{(j)}, y^{(j)}) = \mathbb{P}(y^{(j)}) \cdot \mathbb{P}(x^{(j)} | y^{(j)}) \tag{11.10}$$

Now, what we are aiming to do is to fit a good distribution that will adequately describe the actual data distribution, and we do this by maximizing the joint likelihood function (product of all individual probabilities). From this, we can construct the loss function as the log of this product and we maximize it.

Chapter 12

Hidden Markov Models

Now, we shall look at more sequence learning models for which we are trying to learn some underlying distribution. Specifically, we are going to look at structured prediction problems where we are trying to learn some function that maps one structure to another. Our first approach to doing this will be the hidden Markov model which assumes no memory of prior states apart from the immediate previous state. An important concept to understand hidden Markov models is the naive Bayes assumption. As such, we will first be looking at the naive Bayes model.

§12.1 Naive Bayes

The premise here is that we are going to enforce an independence ('naive') assumption in our model so as to avoid overfitting to a training data set. Let's start with an example. Consider a small data set with a bag of words with $n = 3$ data points (with labels). This data set is written as follows:

x	y
aab	$+1$
ab	$+1$
cdb	-1

Figure 12.1: Simple Bag of Words Data Set

We see there are 2 labels/classes, $+1$ and -1 for each word in the bag of words for which the assigment probabilities to each class is given as $\mathbb{P}(y = +1) = \pi^+$ and $\mathbb{P}(y = -1) = \pi^-$. Let's then look at the first word. The joint probability of the first word and it's label is given as:

$$\begin{aligned}\mathbb{P}(x^{(1)}, y^{(1)}) &= \mathbb{P}(y^{(1)}) \cdot \mathbb{P}(x^{(1)}|y^{(1)}) \\ &= \mathbb{P}(y = +1) \cdot \mathbb{P}(x = \text{aab}|y = +1) \\ &= \mathbb{P}(y = +1) \cdot \mathbb{P}(\text{a}|y = +1) \cdot \mathbb{P}(\text{a}|y = +1, \text{a}) \cdot \mathbb{P}(\text{b}|y = +1, \text{aa})\end{aligned} \tag{12.1}$$

but clearly, if we use such a probability function, we will surely overfit to the training data since we are going to maxmize over the product of all the probability functions of all the words in the

training set! So what we do instead is 'forget' some previous terms. That is to say, we instead use the following probability function:

$$\mathbb{P}(x^{(1)}, y^{(1)}) \approx \mathbb{P}(y = +1) \cdot \mathbb{P}(\mathrm{a}|y = +1) \cdot \mathbb{P}(\mathrm{a}|y = +1) \cdot \mathbb{P}(\mathrm{b}|y = +1) \tag{12.2}$$

so we effectively eliminated the problem of overfitting. This technique of 'forgetting' is known as the *naive Bayes* classifier. As per all other machine learning algorithms, the next step would then be to define a loss function. Note that here, we have a *multinomial distribution* for each $\mathbb{P}(w|y)$ where w indicates the word. So just like a binomial distribution where p is the model parameter (where p is the probability of one event and $1-p$ the other), we have the probability of each event given a classifier outcome $\mathbb{P}(w|y = +/-) = \theta_w^{(+/-)}$ being the model parameters of our multinomial distribution. Our loss function is thus:

$$\begin{aligned}\mathcal{L} &= \mathrm{count}(y = +1) \log \pi^+ + \mathrm{count}(y = -1) \log \pi^- \\ &+ \sum_w \mathrm{count}(w, y = +1) \log \theta_w^+ + \sum_w \mathrm{count}(w, y = -1) \log \theta_w^-\end{aligned} \tag{12.3}$$

with the constraints $\sum_w \theta_w^+ = 1$, $\sum_w \theta_w^- = 1$ and $\pi^+ + \pi^- = 1$. As such, we take the gradient to find the model parameters in the usual means for which we get:

$$\begin{aligned}\pi^+ &= \frac{\mathrm{count}(y = +1)}{\mathrm{count}(y = +1) + \mathrm{count}(y = -1)} \\ \pi^- &= \frac{\mathrm{count}(y = -1)}{\mathrm{count}(y = +1) + \mathrm{count}(y = -1)} \\ \theta_w^+ &= \frac{\mathrm{count}(w, y = +1)}{\sum_w \mathrm{count}(w, y = +1)} \\ \theta_w^- &= \frac{\mathrm{count}(w, y = -1)}{\sum_w \mathrm{count}(w, y = -1)}\end{aligned} \tag{12.4}$$

From this, we can use the results of our maximized model parameters in order to determine the classification of a given word. This is done as follows:

$$\begin{aligned}\log \frac{\mathbb{P}(y = +1|w)}{\mathbb{P}(y = -1|w)} &= \log \frac{\mathbb{P}(y = +1, w)/\mathbb{P}(w)}{\mathbb{P}(y = -1, w)/\mathbb{P}(y = +1, w)} \\ &= \log \frac{\mathbb{P}(y = +1, w)}{\mathbb{P}(y = -1, w)} = \log \frac{\pi^+ \prod_j \theta_{w_j}^+}{\pi^- \prod_j \theta_{w_j}^-}\end{aligned} \tag{12.5}$$

for which we have already found closed form solutions for all the parameters necessary to compute the result above. We took the logarithm of this ratio so we can use the parity (positive or negative) of the result to do the classification. In general, for classification problems of more than 2 possible classes, we instead use the following equation to determine the classification of a given feature:

$$\boxed{\mathrm{argmax}_x\{\mathbb{P}(y|x)\}} \tag{12.6}$$

§12.2 Supervised Hidden Markov Model

Consider a word descriptor classification problem. Given a sentence of words, we want to classify each word (features) x_j into classes y_j where the classes are:

$$y_j \in \{Verb, Determiner, Adjective, Noun\} = \{V, D, A, N\} \tag{12.7}$$

Usually in a sentence, there are certain relations between subsequent words due to grammatical structure (e.g. it is unlikely to have 2 verbs appear adjacent to each other in a sentence). How do we use this to our advantage? In the training phase, we want to maxmize the joint probability distribution:

$$\begin{aligned}\mathbb{P}(x,y) &= \mathbb{P}(x_1,...,x_n,y_1,...,y_n) \\ &= \mathbb{P}(y_1,...,y_5)\cdot\mathbb{P}(x_1,...,x_n|y_1,...,y_n) \\ &= [\mathbb{P}(y_1)\cdot\mathbb{P}(y_2|y_1)\cdot...\cdot\mathbb{P}(y_n|y_1,...,y_{n-1})]\cdot\mathbb{P}(x_1,...,x_n|y_1,...,y_n)\end{aligned} \tag{12.8}$$

Implicit in the construction above, we are saying that we want to generate the classification labels first, and then the features from that. But once again, if we use these conditional probabilities as model parameters, we run into the problem of overfitting to the training data set. So what we can do now is instead to **just** 'remember' the previous word:

$$\mathbb{P}(x,y) \approx [\mathbb{P}(y_1)\cdot\mathbb{P}(y_2|y_1)...\cdot\mathbb{P}(y_n|y_{n-1})]\cdot\mathbb{P}(x_1,...,x_n|y_1,...,y_n) \tag{12.9}$$

One more thing we can add to to improve our model is a 'start' and 'stop' classification label. These are labelled as ('start: y_0) and ('stop': y_{n+1}). As such, this finally gives us our model parameters as follows:

$$\begin{aligned}\mathbb{P}(x,y) &= \mathbb{P}(x_1,...,x_n,y_0,y_1,...,y_n,y_{n+1}) \\ &\approx \left[\prod_{j=1}^{n+1}\mathbb{P}(y_j|y_{j-1})\right]\cdot\mathbb{P}(x_1,...,x_n|y_1,...,y_n)\end{aligned} \tag{12.10}$$

Now, what if wanted to generate words x given some label y? Well, we again get the exact distribution using Bayes' rule and assert some 'forgetfulness' that reduces overfitting:

$$\begin{aligned}\mathbb{P}(x|y) &= \mathbb{P}(x_1,...,x_n|y_0,y_1,...,y_n,y_{n+1}) \\ &= \mathbb{P}(x_1|y_0,y_1,...,y_5,y_6)\cdot...\cdot\mathbb{P}(x_n|y_0,y_1,...,y_5,y_6) \\ &\approx \left[\prod_{j=1}^{n}\mathbb{P}(x_j|y_j)\right]\end{aligned} \tag{12.11}$$

where we have asserted the assumption that each feature (word) is only dependent on the label it is assigned to, which makes sense. With this, the general form of the hidden Markov model for n words and the $n+2$ corresponding labels (with the start and stop labels) is:

$$\boxed{\mathbb{P}(x,y) \approx \left[\prod_{j=1}^{n+1}\mathbb{P}(y_j|y_{j-1})\right]\cdot\left[\prod_{j=1}^{n}\mathbb{P}(x_j|y_j)\right] \approx \left[\prod_{j=1}^{n+1}a_{y_{j-1},y_j}\right]\cdot\left[\prod_{j=1}^{n}b_{y_j(x_j)}\right]} \tag{12.12}$$

where we have defined the quantities a_{y_{j-1},y_j} and $b_{y_j(x_j)}$ which are known as the *transition* and *emission parameters* respectively. In practice, these are given by:

$$a_{y_j,y_{j-1}} = \frac{\text{No. of transitions from } y_{j-1} \text{ to } y_j}{\text{No. of } y_{j-1} \text{ instances}} = \frac{\text{count}(y_{j-1}, y_j)}{\text{count}(y_{j-1})} \tag{12.13}$$

$$b_{y_j(x_j)} = \frac{\text{No. of times } x_j \text{ is generated from } y_j}{\text{No. of } y_j \text{ instances}} = \frac{\text{count}(y_j \to x_j)}{\text{count}(y_j)} \tag{12.14}$$

These transition and emission parameters can be seen as edge weights in a graph. A visualization of this is given in figure 12.2 below:

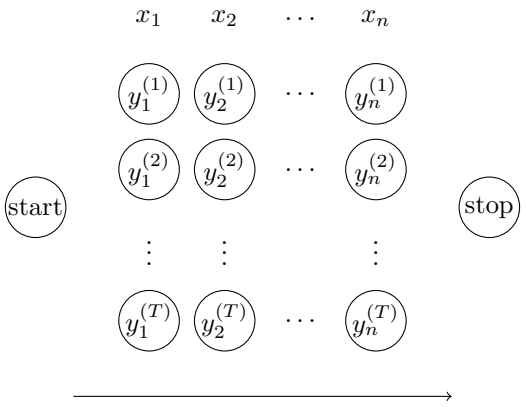

Figure 12.2: Words and Labels Visualization

In figure 12.2, $y_i^{(j)}$ represents the j-th possible label that could be assigned to word (feature) x_i. As mentioned earlier, this would be a fully connected graph where each edge connecting 2 nodes has an edge weight given by:

$$a_{y_{j-1}^{(k)},y_j^{(i)}} \cdot b_{y_j^{(i)}(x_j)} \tag{12.15}$$

where k indicates the k-th label from the previous word layer. (For simplicity, it is easy to just call $y_{j-1}^{(k)} = v$ and $y_j^{(i)} = u$ when looking a each word layer in isolation.) Looking at this, we find that the search space to find the optimal y for each word gets exponentially larger with the number of words ($\mathcal{O}(T^n)$ where n is the number of words and T is the number of possible labels). To reduce the computational complexity of this problem, we perform a *dynamic programming* algorithm called *Viterbi*.

Viterbi Algorithm:

1. We first initialize the start score with a base case:

$$\pi(0, \text{start}) = 1 \tag{12.16}$$

2. At the j-th word (x_j), we assign scores to each possible label u:

$$\pi(j, u) = \max_v \{\pi(j-1, v) \cdot a_{v,u} \cdot b_{u(x_j)}\} \tag{12.17}$$

where v are the possible labels for the $j-1$-th word.

3. Repeat step 2 for $j = 2, ..., n+1$. As such, the final case at the 'stop' layer is:

$$\pi(n+1, \text{stop}) = \max_v \{\pi(n, v) \cdot a_{v,\text{stop}} \cdot 1\} \tag{12.18}$$

Note that these π's are the scores for each label for each word.

4. Finally, we recover the optimal path by backtracking through the word layers and looking for the argmax values for each pi scoring.

The complexity of this algorithm is $\mathcal{O}(nT^2)$, which is a huge improvement over the the originally stipulated $\mathcal{O}(T^n)$. The space complexity is $\mathcal{O}(nT)$ since we require storage of information in every node of the graph.

§12.2.1 Decoding

First, we start with looking at a visualization of the hidden Markov model generative process. This is illustrated in figure 12.3.

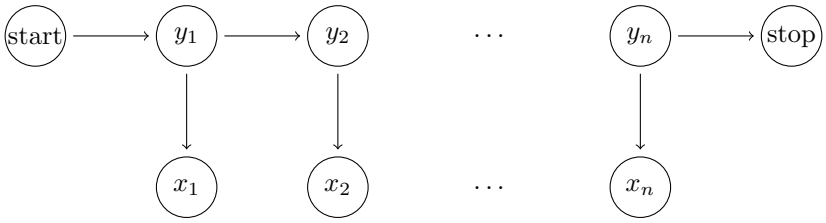

Figure 12.3: Hidden Markov Generative Process

In testing, we once again want to perform $\text{argmax}_x\{\mathbb{P}(y|x)\}$ as per in the naive Bayes (the testing phase is also known as *decoding*). Explicitly, in decoding we are given the features x and model parameters $\theta = \{a_{y_j, y_{j-1}}, b_{y_j(x_j)}\}$ for which we want to find the labels y associated to these features such that:

$$y^* = \text{argmax}_y\{\mathbb{P}(y|x)\} = \text{argmax}_y\{\mathbb{P}(y, x)\} \tag{12.19}$$

We can use the joint probability instead of the conditional probability because the 2 distributions simply differ by a constant prefactor with respect to y.

§12.3 Unsupervised Hidden Markov Model

We now look at a scenario where there are no provided labels associated to the features in our input data. Thus we require to use an **unsupervised** learning algorithm to learn the hidden Markov model. Previously, we have been introduced to expectation maximization (EM) as a means for probabilistic (soft label) clustering of unsupervised data. Loosely speaking, EM was done by iterating the following 2 steps:

1. **E-step**: Finding data membership.
2. **M-step**: Re-estimating model parameters.

We now want to apply this to our context of sequence learning. Let's start first by looking at the case with hard labels because this is a simpler starting point. Initially, we are provided with the features x and some initialized set of model parameters θ, but **not** the labels. What we do here is to first, use our initialized parameters to assign membership (y sequences) to the words in our word sequences (features). This is the M-step. After this, we now have labels to our features, from which we can then use the **supervised** hidden Markov model algorithm above to optimize our model parameters. This is the E-step. We then run these steps iteratively until convergence.

Using this intuition, we now look at the case where we want to adopt soft labels. We are now going to assign the conditional probability of a y sequence given an x feature instead of definite categorizations. As such, the numerators in our transition and emission parameters (12.2) are no longer going to be integer counts, but expected counts ($\in \mathbb{R}$). Note that these expectation values are **local** count information for each feature instance, whereas for the hard label case, those counts encoded **global** count information. These local expected counts can be written as:

$$\begin{aligned}\text{Expected Count of } (u \text{ to } v) &= \mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u, v \text{ in } (X = x, y))\right] \\ &= \sum_{y} \mathbb{P}(y|x) \cdot \text{count}\left(u, v \text{ in } (X = x, y)\right) \\ &= \sum_{y} \mathbb{P}(y|x) \cdot \sum_{j=0}^{n} \text{count}\left(u, v \text{ in } (X = x, y) \text{ at position } j\right)\end{aligned} \tag{12.20}$$

where $\text{count}(u, v \text{ in } (x, y))$ indicates the number of u to v label word transitions in a given y sequence given an x word sequence. This count simply goes over all the possible permutations of label sequences and picks out the number of times we see a u to v transition in all these permutations. Remember that the $\mathbb{P}(y|x)$ are model parameters and the count information is based on either the initialization or the previous iteration of label assignment. Because we are just counting these instances, we can simplify the terms indexed by j with:

$$\text{count}(u, v \text{ in } (x, y) \text{ at position } j) = [[y_j = u, y_{j+1} = v]] \tag{12.21}$$

recalling that $[[...]]$ represents the indicator function (maps to 0 if false or 1 if true). With this, we have that:

$$\begin{aligned}\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u, v \text{ in } (x, y))\right] &= \sum_{j=0}^{n} \sum_{\substack{y \text{ s.t.} \\ y_j = u, \\ y_{j+1} = v}} \mathbb{P}(y|x) \\ &= \sum_{j=0}^{n} \sum_{y_0, \ldots, y_{n+1}} \mathbb{P}(y_0, ..., y_j = u, y_{j+1} = v, ..., y_{n+1}|x) \\ &= \sum_{j=0}^{n} \mathbb{P}(y_j = u, y_{j+1} = v|x)\end{aligned} \tag{12.22}$$

where we have used the identity $\sum_b \mathbb{P}(a,b) = \mathbb{P}(a)$. Using a similar method of derivation as in 12.22, we can also get that:

$$\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u \text{ in } (x,y))\right] = \sum_{j=1}^{n} \mathbb{P}(y_j = u|x) \tag{12.23}$$

Let's look at the each of these probability terms in the sum over j. We want to further decompose them for our understanding and for possible reductions in computational time. We do this as follows:

$$\begin{aligned}
\mathbb{P}(y_j = u|x) &= \mathbb{P}(y_j = u|x_1, ..., x_n) \\
&= \frac{\mathbb{P}(y_j = u, x_1, ..., x_n)}{\mathbb{P}(x_1, ..., x_n)} \\
&= \frac{\mathbb{P}(x_1, ..., x_{j-1}, y_j = u) \cdot \mathbb{P}(x_j, ..., x_n|y_j = u)}{\mathbb{P}(x_1, ..., x_n)} \\
&= \frac{\alpha_u(j) \cdot \beta_u(j)}{\sum_v \alpha_v(k) \cdot \beta_v(k)}
\end{aligned} \tag{12.24}$$

where we used Bayes' rule and also introduced newly defined parameters:

$$\alpha_u(j) = \mathbb{P}(x_1, ..., x_{j-1}, y_j = u) \tag{12.25}$$

$$\beta_u(j) = \mathbb{P}(x_j, ..., x_n|y_j = u) \tag{12.26}$$

This may seem a little strange since there seems to be no restriction on which k we pick for the denominator. The reason for why this is allowed will become clear soon, but we will just take it as true for now since the math works out. We can also do a similar derivation for the pair-sequence $(y_j = u, y_{j+1} = v)$ distributions and construct the individual terms in the sum with α and β:

$$\begin{aligned}
\mathbb{P}(y_j = u, y_{j+1} = v|x) &= \frac{[\mathbb{P}(x_1, ..., x_{j-1}, y_j = u) \cdot \mathbb{P}(x_j|y_j = u) \cdot a_{u,v}] \cdot \mathbb{P}(x_j, ..., x_n|y_j = u)}{\mathbb{P}(x_1, ..., x_n)} \\
&= \frac{\alpha_u(j) \cdot b_{u(x_j)} \cdot a_{u,v} \cdot \beta_u(j)}{\sum_v \alpha_v(k) \cdot \beta_v(k)}
\end{aligned} \tag{12.27}$$

We see that for the u, v transition, we now have to account for the probability that u appears in the j-th position ($\mathbb{P}(x_1, ..., x_{j-1}, y_j = u)$), the emission probability of x_j given u ($\mathbb{P}(x_j|y_j = u)$) and lastly the transition probability for going from u to v ($a_{u,v}$). Lastly, we want to compute the emission probability of some observation (word) o from the label u ($b_u(o)$). To do this, we compute the conditional probability that constitutes the numerator for $b_u(o)$ as follows:

$$\begin{aligned}
\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u \to o)\right] &= \sum_{j=1}^{n} \mathbb{P}(y_j = u, x_j = o|x) \\
&= \sum_{j \text{ s.t. } x_j = o} \frac{\mathbb{P}(x_1, ..., y_j = u, x_j, ..., x_n)}{\mathbb{P}(x_1, ..., x_n)} \\
&= \sum_{j \text{ s.t. } x_j = o} \left(\frac{\alpha_u(j) \cdot \beta_u(j)}{\sum_v \alpha_v(k) \cdot \beta_v(k)} \right)
\end{aligned} \tag{12.28}$$

Great, so we now have all the necessary (expected) count information to generate our transition and emission parameters! This would be done as follows:

$$\tilde{a}_{u,v} = \frac{\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u, v \text{ in } (x,y))\right]}{\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u \text{ in } (x,y))\right]} = \frac{\sum_{j=0}^{n}\left(\frac{\alpha_u(j)\cdot b_{u(x_j)}\cdot a_{u,v}\cdot\beta_u(j)}{\sum_v \alpha_v(k)\cdot\beta_v(k)}\right)}{\sum_{j=1}^{n}\left(\frac{\alpha_u(j)\cdot\beta_u(j)}{\sum_v \alpha_v(k)\cdot\beta_v(k)}\right)} \tag{12.29}$$

$$\tilde{b}_{u(o)} = \frac{\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u \to o)\right]}{\mathrm{E}_{\mathbb{P}(y|x)}\left[\text{count}(u \text{ in } (x,y))\right]} = \frac{\sum_{\{j \text{ s.t. } x_j = o\}}\left(\frac{\alpha_u(j)\cdot\beta_u(j)}{\sum_v \alpha_v(k)\cdot\beta_v(k)}\right)}{\sum_{j=1}^{n}\left(\frac{\alpha_u(j)\cdot\beta_u(j)}{\sum_v \alpha_v(k)\cdot\beta_v(k)}\right)} \tag{12.30}$$

I have used the tilde notation above for these emission and transition parameters since they are being updated during the EM algorithm and do not have deterministic closed form solutions. But now we ask the question, how do we efficiently compute these α and β terms that constitute our unsupervised learning model parameters in practice? First notice that:

$$\mathbb{P}(x_1, ..., x_n) = \sum_{y_0,\ldots,y_{n+1}} \mathbb{P}(x_1, ..., x_n, y_0, ..., y_{n+1}) \tag{12.31}$$

There is a parallel between what we're trying to do here and the Viterbi algorithm. In Viterbi, we were looking at $\max_{y_0,\ldots,y_{n+1}}\{\mathbb{P}(x_1, ..., x_n, y_0, ..., y_{n+1})\}$ whereas now, we are concerned with $\sum_{y_0,\ldots,y_{n+1}} \mathbb{P}(x_1, ..., x_n, y_0, ..., y_{n+1})$. So with the graph picture (12.2) in mind, we see that $\alpha_u(j)$ is in fact just the total probability of all paths entering node u at layer j which begin from the 'start' position. Whereas $\beta_u(j)$ is the total probability of all paths exiting node u at layer j which end at the 'stop' position. This is exactly why choosing an arbitrary k (word) layer did not matter, since the sum over all products of entering and exiting probabilities from any node layer would produce the same result.

Now knowing what these parameters represent, we can compute $\alpha_u(j)$ by simply taking the α values from the previous word layer, multiplying them by their transition parameters $a_{v,u}$ and emission parameters $b_{v(x_{j-1})}$ evaluated in the previous M-step iteration, then summing them up. This is written as:

$$\boxed{\alpha_u(j) = \sum_v \alpha_v(j-1)\cdot b_{v(x_{j-1})}\cdot a_{v,u}} \tag{12.32}$$

In terms of probabilities, this parameter is:

$$\begin{aligned}\alpha_u(j) &= \mathbb{P}(x_1, ..., x_{j-1}, y_j = u) \\ &= \sum_v \mathbb{P}(x_1, ..., x_{j-2}, y_{j-1} = v)\cdot\mathbb{P}(x_{j-1}|y_{j-1} = v)\cdot\mathbb{P}(y_j = u|y_{j-1} = v)\end{aligned} \tag{12.33}$$

For clarity, the base case ('start' to first word layer) is simply:

$$\alpha_u(1) = a_{\text{start},u} \tag{12.34}$$

The remaining α parameters are then computed for $j = 2, 3, ..., n$. As for the $\beta_u(j)$, we can compute these by taking the β values from the next layer, multiply them by their transition

parameters from state u and then sum them up. This is written as:

$$\boxed{\beta_u(j) = \sum_v \beta_v(j+1) \cdot b_{u(x_j)} \cdot a_{u,v}} \tag{12.35}$$

These β parameters are computed for $j = n-1, n-2, ..., 1$ with the $j = n$ base case for this being:

$$\beta_u(n) = a_{u,\text{stop}} \cdot b_{u(x_n)} \tag{12.36}$$

The time complexity of computing all these α and β parameters is exactly the same as computing the scores for the Viterbi algorithm.

§12.3.1 Max-Marginal Decoding

We now look at the method of decoding the unsupervised hidden Markov model. This will be done very similar to how the decoding for a generalized supervised hidden Markov model was done, but instead, we decode over a new parameter space (α and β). After running the unsupervised algorithm for labelling, we have effectively found joint probability distributions of all words and their associated labels:

$$\mathbb{P}(y_0, y_1, ..., y_{n+1} | x_1, ..., x_n) \tag{12.37}$$

As such, we get the optimal labels from this by taking the argmax over the labels y_i such that we maximize this joint distribution. We can do this for word indexed by i and run this over a for loop as follows:

$$\begin{aligned}
&\text{for } i = 1, ..., n: \\
&\quad y_i^* = \arg\max_u \{\mathbb{P}(y_i = u | x_1, ..., x_n)\} \\
&\qquad = \arg\max_u \left\{ \frac{\mathbb{P}(y_i = u, x_1, ..., x_n)}{\mathbb{P}(x_1, ..., x_n)} \right\} \\
&\qquad = \arg\max_u \{\mathbb{P}(x_1, ..., x_{j-1}, y_i = u, x_j, ..., x_n)\} \\
&\qquad = \arg\max_u \{\alpha_u(j) \beta_u(j)\}
\end{aligned} \tag{12.38}$$

This method of decoding is referred to *max-marginal decoding*

Chapter 13

Bayesian Networks

Bayesian networks are generative probabilistic models that were developed for representing and using probabilistic information. All generative models involve variables. How we select values for these variables is governed by a probability distribution. As generative models, Bayesian networks subsume mixture models, hidden Markov models, and many others. In fact, Bayesian networks provide a simple language for specifying generative probability models. There are two parts to any Bayesian network model: 1) a directed acyclic graph over the variables and 2) the associated probability distribution. The graph represents qualitative information about the random variables (conditional independence properties), while the associated probability distribution, consistent with such properties, provides a quantitative description of how the variables relate to each other. The graph structure serves to explicate the properties about the underlying distribution that would otherwise be hard to extract from a given distribution. It also very useful for understanding how we can use the probability models efficiently to evaluate various marginal and conditional properties.

§13.1 Simple Bayesian Networks

Consider the following simple directed acyclic graph (DAG:

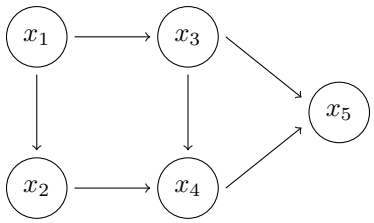

Figure 13.1: 5 Node Directed Acyclic Graph

This as we will see, can represent a generative model. Assuming all x_j can take on binary values, and we want to look at the joint probability $\mathbb{P}(x_1, ..., x_5)$ of the entire graph. On further inspection, we see from the form of the graph that we can factorize its joint probability as

follows:

$$\mathbb{P}(x_1, ..., x_5) = \mathbb{P}(x_1) \cdot \mathbb{P}(x_2|x_2) \cdot \mathbb{P}(x_3|x_1) \cdot \mathbb{P}(x_4|x_2, x_3) \cdot \mathbb{P}(x_5|x_3, x_4) \tag{13.1}$$

This immediately gives us a more explicit picture of the structure of the network if let's say we didn't have figure 13.1 available to us. Since x_j can only take on binary values, we can summarize the probability of each x_j being 0 or 1 in a $2^m \times 2$ matrix, where m is the number of nodes the current node depends on (also known as *parent nodes*). For example, we can represent $\mathbb{P}(x_4|x_2, x_3)$ as:

x_2, x_3	0	1
0 0	0	1
0 1	1	0
1 0	0.7	0.3
1 1	0.1	0.9

Table 13.1: $\mathbb{P}(x_4|x_2, x_3)$ for possible x_4 values

Where the probabilities in table 13.1 above are purely arbitrary and for illustration. Despite them being arbitrarily set, notice that every row in the table/matrix **must** sum to 1 (by the axiom of probability). From here, let us now consider another simpler generative example to better familiarize ourselves with DAGs.

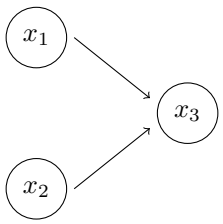

Figure 13.2: 3 Node Directed Acyclic Graph

From figure 13.2 above, we have the joint probability of x_1, x_2 and x_3 to be factorized as:

$$\mathbb{P}(x_1, x_2, x_3) = \mathbb{P}(x_1) \cdot \mathbb{P}(x_2) \cdot \mathbb{P}(x_3|x_1, x_2) \tag{13.2}$$

where these 'factor-ed' terms actually turn our to be our model parameters to be learned. Above, we implicitly assumed that if a node has no parent nodes (no dependencies on previous nodes), then it is statistically independent from all other nodes (assuming no additional information on the graph is given). We can prove this by considering the following:

$$\begin{aligned} \mathbb{P}(x_1, x_2) &= \sum_{x_3} \mathbb{P}(x_1, x_2, x_3) \\ &= \sum_{x_3} \mathbb{P}(x_1) \cdot \mathbb{P}(x_2) \cdot \mathbb{P}(x_3|x_1, x_2) \\ &= \mathbb{P}(x_1) \cdot \mathbb{P}(x_2) \cdot \sum_{x_3} \mathbb{P}(x_3|x_1, x_2) \\ &= \mathbb{P}(x_1) \cdot \mathbb{P}(x_2) \cdot 1 = \mathbb{P}(x_1) \cdot \mathbb{P}(x_2) \end{aligned} \tag{13.3}$$

Which proves that x_1 and x_2 are indeed independent random variables which we can see from the graph.

Note: Notice however that if we are given x_3 (i.e the value of x_3 is known), then x_1 and x_2 are no longer independent!

This method of grouping terms and 'summing them away' is known as *variable elimination*. In a general DAG, variable elimination is an NP-hard problem, but we can come up with approximate methods to doing this as well. Just to make sure we understand what is happening here, let us look at one more example.

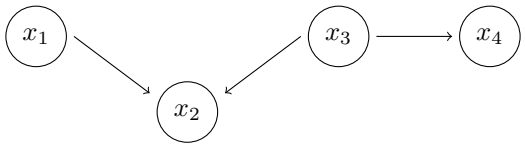

Figure 13.3: 4 Node Directed Acyclic Graph

where $x_j \in \{0, 1\}$ and each of these nodes have the following associated probability tables:

	0	1
	0.01	0.99

Table 13.2: $\mathbb{P}(x_1)$

x_1 x_3	0	1
0 0	1	0
0 1	1	0
1 0	1	0
1 1	0	1

Table 13.3: $\mathbb{P}(x_2|x_1, x_3)$

	0	1
	0.01	0.99

Table 13.4: $\mathbb{P}(x_3)$

x_3	0	1
0	0.3	0.7
1	0.8	0.2

Table 13.5: $\mathbb{P}(x_4|x_3)$

Now, let's say we want to compute $\mathbb{P}(x_1 = 0|x_2 = 0)$. Let us again utilize the method of variable elimination to do this. First we use Bayes' theorem to get that the conditional probability we are concerned with can be also be written as:

$$\mathbb{P}(x_1 = 0|x_2 = 0) = \frac{\mathbb{P}(x_1 = 0, x_2 = 0)}{\sum_{x_1} \mathbb{P}(x_1 = 0, x_2 = 0)} \tag{13.4}$$

Looking at the numerator of the equation above (RHS), we can factorize this as follows:

$$\begin{aligned} \mathbb{P}(x_1 = 0, x_2 = 0) &= \sum_{x_3} \sum_{x_4} \mathbb{P}(x_3) \cdot \mathbb{P}(x_1) \cdot \mathbb{P}(x_4|x_3) \cdot \mathbb{P}(x_2|x_1, x_3) \\ &= \mathbb{P}(x_1) \cdot \sum_{x_3} \mathbb{P}(x_3) \cdot \mathbb{P}(x_2|x_1, x_3) \cdot \sum_{x_4} \mathbb{P}(x_4|x_3) \\ &= \mathbb{P}(x_1) \cdot \sum_{x_3} \mathbb{P}(x_3) \cdot \mathbb{P}(x_2|x_1, x_3) \cdot 1 \end{aligned} \tag{13.5}$$

As such, we simply substitute in the values from the tables to get our result as follows:

$$\begin{aligned}\mathbb{P}(x_1 = 0, x_2 = 0) &= \mathbb{P}(x_1 = 0) \cdot \sum_{x_3} \mathbb{P}(x_2 = 0|x_3) \\ &= 0.01 \times (0.01 \times 1 + 0.99 \times 1) \\ &= 0.01\end{aligned} \tag{13.6}$$

$$\begin{aligned}\mathbb{P}(x_1 = 1, x_2 = 0) &= \mathbb{P}(x_1 = 1) \cdot \sum_{x_3} \mathbb{P}(x_2 = 0|x_3) \\ &= 0.99 \times (0.01 \times 1 + 0.99 \times 0) \\ &= 0.01 \times 0.99\end{aligned} \tag{13.7}$$

$$\begin{aligned}\Rightarrow \quad \mathbb{P}(x_1 = 0|x_2 = 0) &= \frac{\mathbb{P}(x_1 = 0, x_2 = 0)}{\mathbb{P}(x_2 = 0)} \\ &= \frac{0.01}{0.01 + 0.01 \times 0.99} \\ &= \frac{1}{1.99} > 0.5\end{aligned} \tag{13.8}$$

From this, we get that in fact:

$$\mathbb{P}(x_1 = 0|x_2 = 0) \neq \mathbb{P}(x_1 = 0|x_2 = 0, x_3 = 0) \tag{13.9}$$

This implies that x_1 and x_3 are indeed no longer independent once we know the value of x_2. This is quite an interesting observation about information pertaining to conditional and marginal probabilities. This notion is actually known as *explaining away* and will be re-emphasized again soon.

§13.2 Arbitrary Bayesian Networks

So far we have looked at some instances of simple explicit Bayesian networks, but it is always good to construct a generalized model for arbitrarily sized Bayesian networks. As such, we shall extend the intuitions developed from what we have looked at in these simplistic scenarios. We can then extend the exact same inference techniques of variable elimination with the following steps:

Exact Inference for General Bayesian Networks:

For inference problems, we are always looking for the conditional probability of some variable taking on a value, given some other variables taking on other values. This can be written in general as:

$$\mathbb{P}(x_k = x_k^*|x_j = x_j^* \text{ for some set } J \text{ of j values where } j \neq k) \tag{13.10}$$

The algorithm for exact inference can then be performed as followd:

1. First rewrite the conditional probability as in terms of a joint probability using

Bayes' theorem:

$$\mathbb{P}(x_k = x_k^* | x_j = x_j^* \text{ for } j \in J) = \frac{\mathbb{P}(x_k = x_k^*, x_j = x_j^* \text{ for } j \in J)}{\sum_{x_j, j \in J} \mathbb{P}(x_k = x_k^*, x_j \text{ for } j \in J)} \tag{13.11}$$

2. We then consider just the joint probability (numerator) of the entire network and then 'sum away' the unnecessary terms.

$$\mathbb{P}(x_k = x_k^*, x_j \text{ for } j \in J) = \sum_{x_j, j \notin J} \mathbb{P}(x_k = x_k^*, x_j \ \ \forall j \neq k) \tag{13.12}$$

3. We then factorize/decompose the joint probability of the entire Bayesian network using the given model parameters (probabilities in the probability tables) and rearrange the sums such that they only keep relevant terms to their right (This step is NP-hard).

§13.2.1 Model Degrees of Freedom

As earlier mentioned, we saw that the factorized terms in our joint probability constitute the model parameters of our Bayesian network. From our graph visualization of these networks, we can actually see that the total number of model parameters would then be the number of cells in all these probability tables. We now want to find a general formula to count this for any given Bayesian network. This will then allow us to find the number of *degrees of freedom* of our model. Let's start by defining certain variables. Let r_i denote the number of possible values x_i can take on. Then the number of rows and columns in the probability table will be:

$$\text{columns:} \quad r_i \tag{13.13}$$

$$\text{rows:} \quad \prod_{j \in pa} r_j \tag{13.14}$$

where $j \in pa$ means that we only consider the x_j's which are parent nodes of x_i in the product. As such, we have that the total number of parameters is given by:

$$\sum_i \left(r_i \cdot \prod_{j \in pa} r_j \right) \tag{13.15}$$

Great, but remember that in probability theory, probabilities are constrained to always sum to unity. As such, we have that each row in the probability tables for every node must sum to unity, leaving a lower number of possible *free parameters*. As such, the total number of free parameters in any arbitrary Bayesian network model is given by:

$$\boxed{\sum_i \left[(r_i - 1) \cdot \prod_{j \in pa} r_j \right]} \tag{13.16}$$

and we have that the number of free parameters are exactly equal to the number of degrees of freedom of our model. This result will be useful in further analysis of Bayesian networks.

§13.2.2 Independence of Nodes and Bayes' Ball

Earlier, we saw that for the simple Bayesian network 13.3, we could show that 2 originally statistically independent nodes no longer remained statistically independent when a conditional value on some other node was imposed. But now, given some arbitrarily complex Bayesian network, how do we first know if some set of nodes are statistically independent? Furthermore, how do tell now if this set of nodes are still statistically independent when conditioned on given values of some other nodes? First consider again a simple network as visualized in graph 13.4 below.

Figure 13.4: Another Simple Bayesian Network

If we want to check if nodes X and Y are statistically independent, we check if:

$$\mathbb{P}(X,Y) = \mathbb{P}(X) \cdot \mathbb{P}(Y) \tag{13.17}$$

where the statement above being true implies statistical independence. To do this, we once again consider the joint probability:

$$\begin{aligned} & \mathbb{P}(X,Y,Z) = \mathbb{P}(Z) \cdot \mathbb{P}(X|Z) \cdot \mathbb{P}(Y|Z) \\ \Rightarrow \quad & \frac{\mathbb{P}(X,Y,Z)}{\mathbb{P}(Z)} = \mathbb{P}(X|Z) \cdot \mathbb{P}(Y|Z) \\ \Rightarrow \quad & \mathbb{P}(X,Y|Z) = \mathbb{P}(X|Z) \cdot \mathbb{P}(Y|Z) \end{aligned} \tag{13.18}$$

So what we have shown here is that nodes X and Y are statistically independent **only** if the value of node Z is given! The converse (nodes X and Y are dependent if Z is not given) is proven as follows:

	0	1
0	0.56	0.44
1	0.38	0.62

Figure 13.5: $\mathbb{P}(X|Z)$

	0	1
	0.7	0.3

Figure 13.6: $\mathbb{P}(Z)$

	0	1
0	0.67	0.23
1	0.15	0.85

Figure 13.7: $\mathbb{P}(Y|Z)$

Proof. First assume that:

$$\mathbb{P}(X,Y) = \mathbb{P}(X) \cdot \mathbb{P}(Y) \tag{13.19}$$

Now let's consider the case where X, Y and Z can take on 2 values ($\{0,1\}$) each. We then give the probability tables for each node as seen in tables 13.6, 13.5 and 13.7. From these tables, we then compute the RHS and LHS of the equation above to check if it is indeed true.

- Left Hand Side:

To compute these quantities, note that we have:

$$\mathbb{P}(X,Y) = \sum_Z \mathbb{P}(Z) \cdot \mathbb{P}(X|Z) \cdot \mathbb{P}(Y|Z) \tag{13.20}$$

As such, we proceed to compute all possible $\mathbb{P}(X,Y)$ as follows:

$$\begin{aligned}
\mathbb{P}(X=0,Y=0) &= (0.7)(0.56)(0.67) + (0.3)(0.38)(0.15) = 0.27974 \\
\mathbb{P}(X=0,Y=1) &= (0.7)(0.56)(0.23) + (0.3)(0.38)(0.85) = 0.18706 \\
\mathbb{P}(X=1,Y=0) &= (0.7)(0.44)(0.67) + (0.3)(0.62)(0.15) = 0.23426 \\
\mathbb{P}(X=1,Y=1) &= (0.7)(0.44)(0.23) + (0.3)(0.62)(0.85) = 0.22894
\end{aligned} \tag{13.21}$$

- Right Hand Side:
  To compute these quantities, note that we have:

$$\begin{aligned}
\mathbb{P}(X) \cdot \mathbb{P}(Y) &= \left(\sum_Z \mathbb{P}(X,Z)\right) \cdot \left(\sum_Z \mathbb{P}(Y,Z)\right) \\
&= \left(\sum_Z \mathbb{P}(Z) \cdot \mathbb{P}(X|Z)\right) \cdot \left(\sum_Z \mathbb{P}(Z) \cdot \mathbb{P}(Y|Z)\right)
\end{aligned} \tag{13.22}$$

$$\begin{aligned}
&\mathbb{P}(X=0) \cdot \mathbb{P}(Y=0) \\
=&[(0.56)(0.7) + (0.38)(0.3)][(0.67)(0.7) + (0.15)(0.3)] = 0.260084 \\
&\mathbb{P}(X=0) \cdot \mathbb{P}(Y=1) \\
=&[(0.56)(0.7) + (0.38)(0.3)][(0.23)(0.7) + (0.85)(0.3)] = 0.210496 \\
&\mathbb{P}(X=1) \cdot \mathbb{P}(Y=0) \\
=&[(0.44)(0.7) + (0.62)(0.3)][(0.67)(0.7) + (0.15)(0.3)] = 0.253916 \\
&\mathbb{P}(X=1) \cdot \mathbb{P}(Y=1) \\
=&[(0.44)(0.7) + (0.62)(0.3)][(0.23)(0.7) + (0.85)(0.3)] = 0.205504
\end{aligned} \tag{13.23}$$

Comparing the results of the RHS and LHS probabilities, we see that they indeed do **not** equate, and hence our initial assumption is contradicted. □

We can also do this 3-node analysis for the 2 other configurations (both arrows pointing into Z, both arrows flowing in 1 direction), and realize that depending on the configuration of the directed edges, the Z nodes either 'allow' or 'disallow' statistical independence between X and Y (i.e. configuration dictates statistical relationship of X and Y).

With this in mind, we introduce the notion of '*information flow gates*'. The idea is that we want to think of every node with its direction edges between 2 'start' and 'end' nodes in a Bayesian network as gates, and see if those gates allow a 'flow' of probabilistic information from the 'start' to the 'end' nodes. If there exist a continuous flow path, then the start and end notes are statistically **dependent**. They are independent otherwise.

This is likened to a ball rolling through a configuration of pipes from a start to end position, which is why this problem of finding possible edge paths to check statistical independence of

nodes is referred to as *Bayes' ball*. A set of all the different possible gate configurations, their names and their response is given below. Shaded nodes indicate that the values of those nodes are given.

Open Gates

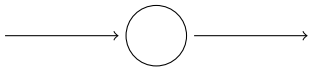

Figure 13.8: Chain Gate

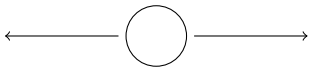

Figure 13.9: Common Cause Gate

Figure 13.10: Explaining Away Gate

Closed Gates

Figure 13.11: Chain Gate (Closed)

Figure 13.12: Common Cause Gate (Closed)

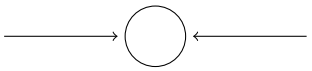

Figure 13.13: Explaining Away Gate (Closed)

It is clear from this formalism that the hidden Markov model is actually just a special case of a Bayesian network.

§13.3 Markov Blankets and Gibb's Sampling

For the Bayesian network problem, so far we have seen to use exact inference methods and variable elimination to get the respectively probabilities. However, when our network becomes arbitrarily large and complex, it becomes computational unfeasible to use these exact methods. As such, we require to utilize approximate methods to get results for our problem. We can simplify the problem by noticing that conditioning on the set of all variables except the state in question is equivalent to conditioning on a few variables surrounding only that state. This set of surrounding nodes/states is called the *Markov blanket*. To be clear, the Markov blanket of a node consists of all the 1) parents, 2) children, and 3) children's parents of that node.

Now, we also want to look at an algorithm that can effectively and efficiently generate the model parameters from the given training data. As such, we will be looking at the Gibb's sampling algorithm which accomplishes this nicely.

Gibb's Sampling Algorithm:
For a Bayesian Network of n nodes, each possibly taking on values x_j and we are given the value of **one** of these nodes x^*. We can generate a table of approximate/predicted values y_j, for each x_j given state values of the Markov blanket of x_j. The algorithm for this is as follows:

1. Randomly initialize $Y^{(0)} = \langle y_1^{(0)}, y_2^{(0)}, ..., y_n^{(0)} \rangle$.

2. for $t = 1, ..., T$:
   (a) for $k = 1, ..., n$:

$$y_k^{(t)} \sim \mathbb{P}(y_k | y_1^{(t)}, ..., y_{k-1}^{(t)}, y_{k+1}^{(t)}, ..., y_n^{(t)}, x^*) \tag{13.24}$$

   (b) Collect the t-th samples $\langle y_1^{(0)}, y_2^{(0)}, ..., y_n^{(0)} \rangle$.
3. Return the samples collected.

§13.4 Supervised Learning in Bayesian Networks

In the supervised learning context, we are given the training set D and the structure (topology) of the Bayesian network. From these, we want to find the model parameters which are the probabilities in the probability tables associated to each node. So for this problem, we again want to maximize the likelihood over the model parameters of observing the given data set. For this, we require the joint probabilities of each feature instance:

$$\mathbb{P}(X_1 = x_1^{(j)}, X_2 = x_2^{(j)}, ..., X_n = x_n^{(j)}) \tag{13.25}$$

where the superscript (j) indicates that this is the value from the j-th observed feature. Since we know the network structure, we can further decompose each of these observed joint probabilities into smaller terms which are conditional probabilities. Each of these can then be found using count information from the data set. The probabilities for every possible value of a given node given the possible values of its parent nodes can be put into a table/matrix. To get a clearer picture of this, let's look at an example. Consider a simple 3 node Bayesian network.

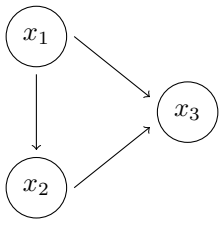

Figure 13.14: 3 Node Example

If x_j can take on r_j different possible values ($j \in \{1, 2, 3\}$), then the matrix of probabilities for node x_3 is given by:

x_1	x_2	1	2	...	r_3
1	1	$\theta_3(1\|1,1)$	$\theta_3(2\|1,1)$	...	$\theta_3(r_3\|1,1)$
⋮	⋮	⋮	⋮		⋮
1	r_2	$\theta_3(1\|1,r_2)$	$\theta_3(2\|1,r_2)$	...	$\theta_3(r_3\|1,r_2)$
⋮	⋮	⋮	⋮	⋱	⋮
r_1	1	$\theta_3(1\|r_1,1)$	$\theta_3(2\|r_1,1)$	...	$\theta_3(r_3\|r_1,1)$
⋮	⋮	⋮	⋮		⋮
r_1	r_2	$\theta_3(1\|r_1,r_2)$	$\theta_3(2\|r_1,r_2)$	...	$\theta_3(r_3\|r_1,r_2)$

Figure 13.15: Probability Table for Node x_3

Where in table 13.15 above, each of the conditional probabilities are model parameters. So now that we have this picture of what our model parameters are representing, we can generalize this idea for an arbitrary Bayesian network given m observed feature instances and a Bayesian network of n states/nodes. The total likelihood function of observed features is defined as the product of joint probability distributions for each feature instance given by:

$$\prod_{t=1}^{m} \mathbb{P}(X_1 = x_1^{(t)}, X_2 = x_2^{(t)}, ..., X_n = x_n^{(t)}) \tag{13.26}$$

As before, we want to take the log of the likelihood function that maximize that. But since we know the Bayesian network structure, this can be written as:

$$\begin{aligned}
\log\left[\prod_{t=1}^{m} \mathbb{P}(X_1 = x_1^{(t)}, X_2 = x_2^{(t)}, ..., X_n = x_n^{(t)})\right] &= \log\left[\prod_{t=1}^{m}\left(\prod_{i=1}^{n} \mathbb{P}(X_i = x_i^{(t)} | X_{pa(i)} = x_{pa(i)}^{(t)})\right)\right] \\
&= \sum_{t=1}^{m}\sum_{i=1}^{n} \log \mathbb{P}(X_i = x_i^{(t)} | X_{pa(i)} = x_{pa(i)}^{(t)}) \\
&= \sum_{i=1}^{n}\left[\sum_{t=1}^{m} \log \theta(x_i^{(t)} | x_{pa(i)}^{(t)})\right]
\end{aligned} \tag{13.27}$$

where the subscript $pa(i)$ indicates all parent nodes of node i. We then train the model by finding the optimal parameters via minimization (gradient descent) of the negative log-likelihood (loss function) over the $\sum_i \left[(r_i - 1)\cdot\prod_{j\in pa} r_j\right]$ parameters. For clarity, let me restate the loss function associated to this learning supervised problem.

$$\boxed{\mathcal{L}(\theta; S_m) = -\sum_{i=1}^{n}\left[\sum_{t=1}^{m} \log \theta(x_i^{(t)} | x_{pa(i)}^{(t)})\right]} \tag{13.28}$$

§13.5 Structure Learning in Bayesian Networks

Now we move to a different learning paradigm where we are instead **only** given a training set D, and we are supposed to find both the structure of the Bayesian network and the model

parameters. Let's first consider a situation of just 2 nodes, each of which can take on 2 possible values ($\{0, 1\}$) and say we are given the following data set.

X	Y
0	0
0	1
1	0
1	1
1	0
0	1
0	0

Figure 13.16: 2 Node Network Data Set

There 2 nodes can then have 3 possible configurations as acyclic graphs, for which the associated probability tables (using count information) are given as follows:

0	1
4/7	3/7

Figure 13.17: State X in G_1

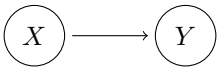

Figure 13.20: G_1

	0	1
0	2/4	2/4
1	2/3	1/3

Figure 13.23: State Y in G_1

	0	1
0	2/4	2/4
1	2/3	1/3

Figure 13.18: State X in G_2

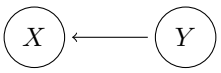

Figure 13.21: G_2

0	1
4/7	3/7

Figure 13.24: State Y in G_2

0	1
4/7	3/7

Figure 13.19: State X in G_0

Figure 13.22: G_0

0	1
4/7	3/7

Figure 13.25: State Y in G_0

From here, we propose the use of log-likelihood maximization in order to find an optimal structure and model parameters. However, the issue with just using log-likelihood maxmization is that it does not give any preferencing between the configurations G_1 and G_2. Furthermore, it will always assign a lower score to independent nodes (by the law of total probability), causing the complexity of the output graph to be high. The problem we are facing here is essentially model selection, and so to deal with the issue of an over complex model, we utilize the Bayesian information criterion (BIC). As such, we modify loss function to:

$$\boxed{\mathcal{L}(\theta; S_m) = \sum_{i=1}^{n} \left[\sum_{t=1}^{m} \log \theta(x_i^{(t)} | x_{pa(i)}^{(t)}) - \frac{\log(m)}{2}(r_i - 1) \prod_{j \in pa} r_j \right]} \tag{13.29}$$

Chapter 14

Reinforcement Learning

We now want to introduce a whole new paradigm of learning that does not really fall under the conventional supervised and unsupervised learning categories. Before we get to describing this new form of learning, we once again stress that in machine learning, we are doing nothing but learning functions. The function that we now want to learn in this chapter is written as:

$$\pi : S \to A$$

where S is a set of states, and A is a set of actions. This function is referred to as the ***policy****, and it tells a machine what action a to take given that the machine exists in some state s. As such, we can train this policy function by asking the machine to take an action at each state and then giving a response (feedback) to it. This constitutes what is known as* ***reinforcement learning****. We look first at an introductory example where learning such a function would be useful.*

§14.1 Robot Path Learning

Consider the following problem. Let's say there is a robot trying to find its way through a room (possibly with obstacles) towards an exit. How do we construct a training protocol to teach the robot how to move through this room? Well a good first step would be to discretize the room into cells (adopting some grid structure). Earlier, we mentioned that we would like a regime in which we provide some form of feedback to the robot during its training (just as a new dog owner would do to train his puppy). With this in mind, a viable means of training would be to design a 'reward system' by assigning each cell in the grid with a *score*. Now comes the tricky part, how do we design a function related to these scores for the robot to optimize such that it can find the best possible path to the exit?

Let's start with some abstractions. We want the robot to try and accumulate the highest possible score by taking the 'right' steps, but at the same time, we also don't want the robot to simply go in circles around the room so that it can keep accumulating a higher score indefinitely. As such, there are 2 considerations in designing the robot's objective function:

1. The objective function should reward the robot if it takes actions with a higher scores.

2. The objective function should penalize the robot for taking too many steps.

Let's call this function to be optimized the *utility* U (analogous to utility maximization of consumers in economic theory). With the 2 considerations above, we can then define this utility function over some given path of n steps as:

$$U(s_1, s_2, ..., s_n) = \sum_{j=1}^{n} R(s_j) \cdot \alpha^{j-1} \tag{14.1}$$

where s_j denotes the states over some path of n steps and $\alpha \in (0, 1)$ is some discount factor that lowers the score of future steps which innately prevents over complexity of the path. So we see that this elegant function addresses both considerations! However, let's just make sure that this utility function is upper bounded so it will always attain a finite value no matter the number of path steps ($n \to \infty$):

$$\begin{aligned}
\sum_{j=1}^{\infty} R(s_j) \cdot \alpha^{j-1} &\leq \sum_{j=1}^{\infty} \max_j\{R(s_j)\} \cdot \alpha^{j-1} \\
&= \max_j\{R(s_j)\} \cdot \sum_{j=1}^{\infty} \alpha^{j-1} \qquad (14.2) \\
&= \max_j\{R(s_j)\} \cdot \left(\frac{1}{1-\alpha}\right) \\
\Rightarrow \quad &\boxed{U(s_1, s_2, ..., s_n) \leq \max_j\{R(s_j)\} \cdot \left(\frac{1}{1-\alpha}\right)} \qquad (14.3)
\end{aligned}$$

So our utility function is indeed upper-bounded. Let us also be aware that robots are hardware devices, as such, there will be some uncertainties in their actual paths traversed vs what the software has intended for them. These must be taken into account, and we can model these as *transition probabilities* of moving from its current state s to the next state s' (or other surrounding viable states), given the intended action a of moving to some new state ($\mathbb{P}(s'|s, a)$). Let's make a list of the parameters we have so far:

1. A set of states, $\{s_i\}$.
2. A set of actions, $\{a_i\}$.
3. The reward function $R(s, a, s')$.
4. Transition probabilities $T(s, a, s') = \mathbb{P}(s'|s, a)$

These parameters actually constitute what is known as a *Markov decision process (MDP)*, and are the assumed 'givens' in general reinforcement learning problems. With this, we now want to get a procedure to learn the optimal policy function (π^*) such that it maximize the robot's utility U. Let's recap and define a couple more quantities that will aid us in how to go about doing this.

1. $\pi(s)$: the action taken at state s.
2. $\pi^*(s)$: the optimal action to take at state s.
3. $V^{\pi}(s)$: the expected utility for a given state s by following a policy π.
4. $Q^{\pi}(s, a)$: the expected utility for the state s after taking an action a, and subsequently following a policy π thereafter.

We can then ask, what is the relation between the $Q^{\pi}(s,a)$, $V^{\pi}(s)$ and $\pi(s)$? First, consider the optimal path $\pi^*(s)$. Then we actually see that:

$$\pi^*(s) = \arg\max_a \{Q^{\pi^*}(s,a)\} \tag{14.4}$$

$$V^{\pi^*}(s) = \max_a \{Q^{\pi^*}(s,a)\} = Q^{\pi^*}(s, \pi^*(s)) \tag{14.5}$$

$$Q^{\pi^*}(s,a) = \sum_{s'} T(s,a,s') \cdot \left[\alpha V^{\pi^*}(s') + R(s,a,s')\right] \tag{14.6}$$

$$\Rightarrow \boxed{V^{\pi^*}(s) = \max_a \left\{ \sum_{s'} T(s,a,s') \cdot \left[\alpha V^{\pi^*}(s') + R(s,a,s')\right] \right\}} \tag{14.7}$$

Make sure that you understand why the equalities above (14.4, 14.5, 14.6) hold.

§14.1.1 Value Iteration Algorithms

From the boxed equation above, we can in fact already construct our protocol! We call this the *value iteration* protocol which is presented below.

Value Iteration Protocol:

1. Initialize $V_0^*(s) = 0$ for all s.
2. Compute $V_{i+1}^*(s')$ based on $V_i^*(s)$ for all s:
   (a)
   $$V_{i+1}^*(s) \leftarrow \max_a \left\{ \sum_{s'} T(s,a,s') \cdot [\alpha V^*(s')_i + R(s,a,s')] \right\} \tag{14.8}$$
   (b) At this step, we also compute
   $$\arg\max_a \left\{ \sum_{s'} T(s,a,s') \cdot [\alpha V^*(s')_i + R(s,a,s')] \right\} \tag{14.9}$$
   and store these indices (grid positions) so that we generate the optimal path through this algorithm.
3. Repeat step 2 until convergence.

In the protocol above, we have used 'lighter' notation by replacing π^* as * in the superscripts of V_j. Let's now ask the question, is there a way we can directly compute the values of Q, therefore bypassing the need to compute V? It turns out that it is indeed possible and in fact, largely similar to the way we found an equation purely involving V. This is done as follows:

$$\pi^*(s) = \arg\max_a \{Q^{\pi^*}(s,a)\} \tag{14.10}$$

$$V^{\pi^*}(s) = \max_a \{Q^{\pi^*}(s,a)\} = Q^{\pi^*}(s, \pi^*(s)) \tag{14.11}$$

$$Q^{\pi^*}(s,a) = \sum_{s'} T(s,a,s') \cdot \left[\alpha V^{\pi^*}(s') + R(s,a,s')\right] \tag{14.12}$$

$$\Rightarrow \boxed{Q^{\pi^*}(s,a) = \sum_{s'} T(s,a,s') \cdot \left[R(s,a,s') + \alpha \max_{a'} \{Q^{\pi^*}(s',a')\}\right]} \tag{14.13}$$

With this, we can define a new value iteration procedure for Q which we call the Q-value iteration algorithm.

Q-Value Iteration Protocol:

1. Initialize $Q_0^*(s) = 0$ for all (s,a).
2. Compute $Q_{i+1}^*(s',a')$ based on $Q_i^*(s,a)$ for all s:
   (a)

$$Q_{i+1}^{\pi^*}(s,a) \leftarrow \left\{\sum_{s'} T(s,a,s') \cdot \left[R(s,a,s') + \alpha \max_{a'} \{Q_i^{\pi^*}(s',a')\}\right]\right\} \tag{14.14}$$

   (b) At this step, we also compute

$$\arg\max_a \left\{\sum_{s'} T(s,a,s') \cdot \left[\alpha \max_{a'} \{Q_i^{\pi^*}(s',a')\} + R(s,a,s')\right]\right\} \tag{14.15}$$

   and store these indices (grid positions) so that we generate the optimal path through this algorithm.
3. Repeat step 2 until convergence.

Finally, can we also construct a 'policy iteration protocol'? Let's first consider the equation:

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s') \cdot [\alpha V^{\pi}(s') + R(s,a,s')] \tag{14.16}$$

We want to make an equation that **only** involves V, as such, we can replace all a's with $\pi(s)$ to get:

$$Q^{\pi}(s,\pi(s)) = V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s') \cdot [\alpha V^{\pi}(s') + R(s,\pi(s),s')] \tag{14.17}$$

From here, we can establish a system of linear equations for the N different s states. We can then use some linear solver to find the solutions for the optimal policy. However, this algorithm is much more computationally expensive than the previous 2 value iteration algorithms (since Gaussian elimination has a high time complexity).

§14.2 General Reinforcement Learning Scheme

In a generalized reinforcement learning problem, we do not know that reward function R, or the transition probabilities T. As such, we are only given the following to start with:

1. A set of states.
2. A set of actions.

For our machine to learn here, we can instead take a model-free approach. First consider the following game.

Coin Toss Game:
There is a game master who flips a coin. If the result is heads, we get \$20, but if the result of the flip is tails, we have to pay the game master \$10. Would you play this game? What we aim to do here is to find $\mathbb{P}(H)$ before we make a decision to play or not. How we can do this is by first observing other players and their outcomes. Let's say we observe another player who gets the following results:

$$\{H, H, T, H, T, T, T, T, T, T\} \tag{14.18}$$

We can update our belief about the game at every new observed instance of the game result with the following equation:

$$\mathrm{E}_k[\text{money gained}] = \frac{v \cdot (k-1) + (t)}{k} \tag{14.19}$$

where k is the index of the current game round, v is the expected monetary gain in the $k-1$th round, and t is the money gain outcome in round k.

Extending the intuition gained from the example game to our robot learning context, what we can do is come up with the expected score values at each time step as the robot collects more data samples over time.

$$Q_{new}(s,a) = \frac{Q_{old}(s,a) \cdot (k-1) + [R(s,a,s') + \alpha \cdot \max_{a'}\{Q(s',a')\}]}{k} \tag{14.20}$$

From this, we can construct an algorithm to perform this dynamic Q-learning problem.

Q-Learning:
1. Collect a sample (s, a, s')
2. Update $Q(s,a)$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{k}\Big[R(s,a,s') + \alpha \cdot \max_{a'}\{Q(s',a')\} - Q(s,a)\Big] \tag{14.21}$$

3. Repeat step 2 until convergence.

Notice that the update equation looks very much like the gradient descent update sequence where $\frac{1}{k}$ is simply the learning rate, previously denoted as η. There is actually a deep connection between the learning rate here and that for the gradient descent, but this will not be covered in

this course. A concern with this algorithm is that, if we don't explore a large region of the world, our robot's optimal policy could get restricted to a local environment. Conversely, exploring a large region of the world could be very time consuming. As such, there is a trade-off in optimizing between these 2 considerations. The solution to this has to be found via empirical means.

Appendices

Appendix A

Lagrangian Dual Problem

Here, we run through the definition of Lagrangians and the mathematical reasoning as to why we utilize the dual problem to solve a given constrained optimization problem. First, we define what a *Lagrangian* is.

Definition A.0.1. Lagrangian: *Given the optimization problem $(*)$ defined by some function $f(x)$ to be optimized and subject to some constraints :*

$$\min \quad \{f(x)\} \tag{A.1}$$

$$\begin{aligned} \text{Subject to:} \quad & h_i(x) = 0, \quad 1 \leq i \leq l \\ & g_j(x) \leq 0, \quad 1 \leq j \leq m \end{aligned} \tag{A.2}$$

We define the Lagrangian associated to $(*)$ as:

$$\begin{aligned} L : D \times \mathbb{R}^l \times \mathbb{R}^m &\to \mathbb{R} \\ (x, \lambda, \alpha) &\mapsto f(x) + \sum_{i=1}^{l} \lambda_i h_i(x) + \sum_{j=1}^{m} \alpha_j g_j(x) \end{aligned} \tag{A.3}$$

where D is the domain on which $f(x)$ is defined on (i.e. $x \in D \subset \mathbb{R}^n$).

Notice the form of the constraints we have written up. This is the convention and is essential that we adhere to this convention when constructing our Lagrangian to obtain an optimal result. From here, we can construct a *Lagrangian dual function* which provides a new optimization problem as follows:

Definition A.0.2. Lagrangian Dual Function: *Given the optimization $(*)$ and its associated Lagrangian $L(x, \lambda, \alpha)$, we define the corresponding Lagrangian dual function as:*

$$g : \mathbb{R}^l \times \mathbb{R}^m \to \mathbb{R} \tag{A.4}$$

$$(\lambda, \alpha) \mapsto \min_{x \in D} \big\{ L(x, \lambda, \alpha) \big\} \tag{A.5}$$

We can think of $L(x, \lambda, \alpha)$ as a family of functions that depend on λ and α but indexed by x, $L_x(\lambda, \alpha)$. With this perspective, we see that $L_x(\lambda, \alpha)$ is an affine function in λ and α:

$$L_x(\lambda, \alpha) = f + \sum_{i=1}^{l} \lambda_i h_i + \sum_{j=1}^{m} \alpha_j g_j \tag{A.6}$$

So we are essentially treating f, h_i and g_j as constants (w.r.t to λ and α).