Machine Learning in Science Part 1 (PHYS4035) Lecture Notes

Juan P. Garrahan, Edward Gillman and Jamie F. Mair

September 26, 2025

Contents

Introduction

These are the lecture notes for the module Machine Learning in Science Part 1 (PHYS4035) of the Machine Learning in Science MSc (often abbreviated as MLiS MSc) taught at the University of Nottingham.

The module has three parts: supervised learning (SL), unsupervised learning (USL) and reinforcement learning (RL).

All material for MLiS part 1 is found in the module's Moodle page. For general references see the Bibliography at the bottom of this document.

Part I

Supervised Learning

Resources

Linear Algebra:

1. Chapter 3 of Mathematical Methods in the Physical Sciences 3rd Edition, Mary L. Boas [1]
2. Chapter 2 of Deep Learning, Goodfellow, Bengio and Courville [3].

Partial Differentiation:

1. Chapter 4 of Mathematical Methods in the Physical Sciences 3rd Edition, Mary L. Boas [1]

Chapter 1

Basic Ingredients for Supervised Learning

1.1 Tasks, Performance Measures and Experience

A basic definition of machine learning (ML) is given by Mitchell (1997): "A computer program is said to *learn* from experience, E, with respect to some task, T, as measured by a performance measure, PM, if the PM improves with E" [3].

Some basic examples of tasks, performance measures and experience, along with related definitions, in the context of supervised learning are:

Tasks (T)

- Eg. 1: *Regression* – Predict the numerical value of some quantity, given a corresponding example input. For example, predict the extent of Arctic ice, given the year.
- Eg. 2: *Classification* – Specify to which category a given input corresponds. For example, the condition of vegetation (healthy, damaged or devastated) about a volcano, given the concentration of SO_2.
- Note 1: More generally, we might also want to quantify the uncertainty on such predictions, i.e., also provide probabilities for the associated values/categories.
- Note 2: In machine learning tasks, we are typically interested in making predictions for previously unseen cases, rather than simply "memorising" known cases.

Performance Measures (PM)

- Eg. 1: *Mean Squared Error* – Used in regression tasks, this is the average squared difference between our predicted values and the actual measured values, given some set of inputs.
- Eg. 2: *Accuracy* – The proportion of correctly classified examples on some chosen set of inputs.
- Note: We must choose which examples to calculate our performance measure over. If we choose examples that have already been seen and learnt from (these are the examples found in E) then performance is likely to be much better than on unseen examples. This second case, often called "generalisation" is typically a better measure of the true task we are interested in.

Experience (E)

- Data: The set of examples that have been measured, i.e., the data points. For example, the data (or dataset) might consist of records of the extent of Arctic ice along with the year, or measurements of the condition of vegetation and concentration of SO_2.

- Supervised Learning (SL): In these problems, the data is divided clearly into the "target" quantity (to be predicted in the T) and the "input" quantity. Note, this is the case in both example Ts above.
- Note 1: In unsupervised learning (UL) the data contains no clear "target" to be predicted.
- Note 2: There is no clear-cut division between SL and UL, and SL problems can be phrased as UL problems and vice verse.

1.2 Some Mathematical Notation for Supervised Learning

To understand the structure of supervised learning methods, we must quantify the general concepts of T, PM and E that provide a conceptual framework for machine learning. To do this, we being by introducing some mathematical notation. For now, this will be restricted to the case of deterministic machine learning models, and we will generalise it as needed when probabilistic machine learning models are discussed later on.

1.2.1 Mathematical Notation for E

The experience, E, will be encoded in a dataset, $\mathcal{D}$. This consists of N examples, indexed by $i = 1, 2, ..., N$. In SL, each example is denoted as $z^{(i)}$ and consists of a pair $z^{(i)} = \left(x^{(i)}, y^{(i)}\right)$. Here, $y^{(i)}$ is the "target" in the ith example, while $x^{(i)}$ is the "input", which we will also call a "feature" interchangeably. Generally, there will be multiple input quantities (e.g. the year, the amount of atmospheric CO_2 etc.) or multiple targets (e.g. for multiple categories). In these cases we write, $\mathbf{z}^{(i)} = \left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$, where the bold indicates the quantities are vectors, comprised of several quantities. We will denote the number of input quantities P, and label the individual quantities by $p = 1, 2, ..., P$. The components of $\mathbf{x}^{(i)}$ are therefore denoted as $x_p^{(i)}$ (we will use subscript indices for the components of vectors and matrices generally) so that $\mathbf{x}^{(i)} = (x_1, x_2, ..., x_P)^T$. When we have multiple target variables, we will denote he number of outputs as K, so that the dimension of $\mathbf{y}^{(i)}$ would be K, and components are denoted as $y_k^{(i)}$ with $k = 1, 2, ..., K$.

A useful tool for encoding the features of $\mathcal{D}$ is the design matrix, $\mathbf{X}$. The entries (components), X_{ip}, of $\mathbf{X}$ are the values of pth feature in the ith examples input, $X_{ip} = x_p^{(i)}$. In other words, the rows of $\mathbf{X}$ are input vectors from the dataset. Often – we will see why shortly – it is helpful to also have a first "artificial" column in $\mathbf{X}$ of all ones, which we label by $p = 0$. So, the final shape of $\mathbf{X}$ will be $(N, P + 1)$ and its total size (i.e. the total number of components) will be $N \times (P + 1)$, where $X_{i0} = 1$ for all examples. Typically, we will also extend this notation to other objects, e.g. $\mathbf{x}^{(i)}$, so that the zeroth component is one by definition.

Corresponding to the design matrix for the inputs, we can also encode the targets of our data set in a matrix, $\mathbf{Y}$, which then has components, $Y_{ik} = y_k^{(i)}$. The shape of $\mathbf{Y}$ is then (N, K), and it has $N \times K$ entries in total. When $K = 1$, as in many cases we will look at, this is just a vector of targets from $\mathcal{D}$, i.e., the components are $Y_i = y^{(i)}$. We will refer to $\mathbf{Y}$ as the target matrix, or target vector when $K = 1$.

In summary, in our SL problems, the experience E will be encoded in a data set, $\mathcal{D}$, of N examples or, equivalently, the design and target matrices, $\mathbf{X}$ and $\mathbf{Y}$, with shapes $(N, P + 1)$ and (N, K) respectively.

1.2.2 Mathematical Notation for T

We will be interested in making predictions of the target quantity that can be derived uniquely – and mathematically – from any given input. We will denote a generic example input as x or $\mathbf{x}$. Note that this is not necessarily taken from $\mathcal{D}$, as it could be previously unseen.

Predictions made by our models will be indicated by a hat. Depending on the task in hand, we will want to predict different things. In the simplest case, on which we will focus at first, we want to predict a target quantity, $\mathbf{y}$, corresponding to some single input, $\mathbf{x}$. As we only predict an output and do not quantify any uncertainty in this prediction, we will call such a prediction (along with the corresponding model) deterministic.

The actual prediction we make with our model is then denoted as $\hat{\mathbf{y}}$. Since we would like to make this prediction uniquely, the prediction should be a function of the input. In other words, specifying our

machine learning model corresponds to specifying some function f. In the case that $K = 1$ and $P = 1$, we simply have,

$$\hat{y} = f(x) \ . \tag{1.1}$$

When $P > 1$, we generalise this to,

$$\hat{y} = f(\mathbf{x}) \ , \tag{1.2}$$
$$= f(x_1, x_2, ..., x_P) \ , \tag{1.3}$$

so that the function takes P input values. Finally, when $K > 1$, we have,

$$\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}) \tag{1.4}$$
$$= (f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_K(\mathbf{x}))^T \ , \tag{1.5}$$

so that now the function, $\mathbf{f}$, is vector-valued, i.e., it outputs a set of K values, corresponding to K different functions arranged as a vector, rather than just a single number.

Finally, as we are often interested in seeing what our model predicts specifically for the set of examples contained in $\mathcal{D}$, we can introduce the prediction matrix, $\hat{\mathbf{Y}}$. This matrix, of shape (N, K), has components $\hat{Y}_{ik} = \hat{y}_k^{(i)}$, i.e., the predictions corresponding to each target variable (labelled by k) for each example in $\mathcal{D}$ (labelled by i). In the case that $K = 1$, this is just a vector of predictions corresponding to the N target variables contained in $\mathcal{D}$.

In summary, for a given T, we will construct a model to perform T. This will produce predictions appropriate to the T, which we denote with a hat. In the cases we will look at first, the predictions will be of the target variable without uncertainty, so that $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x})$ in the most general case.

1.2.3 Mathematical Notation for PM

Finally, for deterministic models, performance will be quantified on a particular example, $\mathbf{z} = (\mathbf{x}, \mathbf{y})$, by comparing the (true) measured value of the target with the predicted value given by our model.

Typically, we divide performance measures into different categories. The most important of these for now is the loss. This is the performance measure we single out and directly aim to improve when developing the mathematical procedure behind our machine learning method. Later, we will see how we can find general principles to help us determine what form the loss takes for a given task. Other performance measures, that we might hope to improve but do not directly optimise, we will call metrics, and will return to later on.

Given a particular example $(\mathbf{x}, \mathbf{y})$, we will denote the *per-example-loss* as l. Note that this will be a function of both the target, $\mathbf{y}$, and the prediction, $\hat{\mathbf{y}}$, i.e., $l(\hat{\mathbf{y}}, \mathbf{y})$.

When we consider the loss on multiple examples, we will denote it as L. This can be calculated by summing or averaging the per-example-loss on a set of chosen examples. The most important case of this is that of averaging over the examples contained in $\mathcal{D}$. This is because is it the loss calculated on $\mathcal{D}$ that we will aim to improve directly in our machine learning method.

Therefore, the loss over $\mathcal{D}$ can be written as,

$$L_{\mathcal{D}} = \frac{1}{N} \sum_{i=1}^{N} l(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \ . \tag{1.6}$$

Later, we will also be interested in evaluating performance measures (both the loss and metrics) on examples from outside $\mathcal{D}$. However, for the mathematical development of our algorithms, we must first focus on the loss calculated on $\mathcal{D}$, for reasons that will become clear shortly.

In summary, the PM we wish to improve directly can be quantified by specifying $L_{\mathcal{D}}$. It is important to note that the specification of $L_{\mathcal{D}}$ doesn't depend on how the predictions are made, and so is independent of the chosen model.

Example 1: T, PM and E for Arctic Ice Extent

In these notes we will look at two example tasks for SL. The first concerns the extent of Arctic ice. The dataset encoding E we will look at for now is the set of year-averaged Arctic ice extents (the basic data set for this example is available from the National Snow and Ice Data Center, https://nsidc.org/). We will look at data for the years between and including 1979 and 2018. Therefore, $N = 40$. The first few data points are:

$$\mathcal{D} = \{(1979, 12.44), (1980, 12.29), (1981, 12.2), ..., (2018, 10.34)\} \ . \tag{1.7}$$

Note that for this dataset, $P = K = 1$.

The target vector, $\mathbf{Y}$, is of shape $(N, 1)$, and written as,

$$\mathbf{Y} = \begin{pmatrix} 12.44 \\ 12.29 \\ ... \\ 10.34 \end{pmatrix} \ , \tag{1.8}$$

while the design matrix, $\mathbf{X}$, is of size 40×2, and given by;

$$\mathbf{X} = \begin{pmatrix} 1 & 1979 \\ 1 & 1980 \\ ... & ... \\ 1 & 2018 \end{pmatrix} \ . \tag{1.9}$$

To extend this data set to more target variables ($K > 1$) we could, e.g., consider predicting the ice extent for each month so that $K = 12$. To increase the number of features, we could additionally consider the global mean temperatures and greenhouse gas concentrations, in which case $P = 3$.

A basic task we might be interested in is predicting the extent of Arctic ice, y, given a year, x. Since the target Arctic ice extent is a numerical quantity, this is a regression task. Furthermore, as there is one input quantity and one target quantity, for a deterministic SL model our task boils down to choosing the function f, where, $\hat{y} = f(x)$.

A very simple model, which we can just design by hand, would be to take the mean Arctic ice extents in our data set, and simply predict this regardless of the input year. In other words, our model would be specified as,

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} y^{(i)} \ . \tag{1.10}$$

Of course, this is not a function of x at all. However, it is still an important model – which we will call the constant model – as it encodes an assumption of independence between the input and target variables.

The data $\mathcal{D}$, along with the predictions of the constant model, are plotted in Fig 1.1.

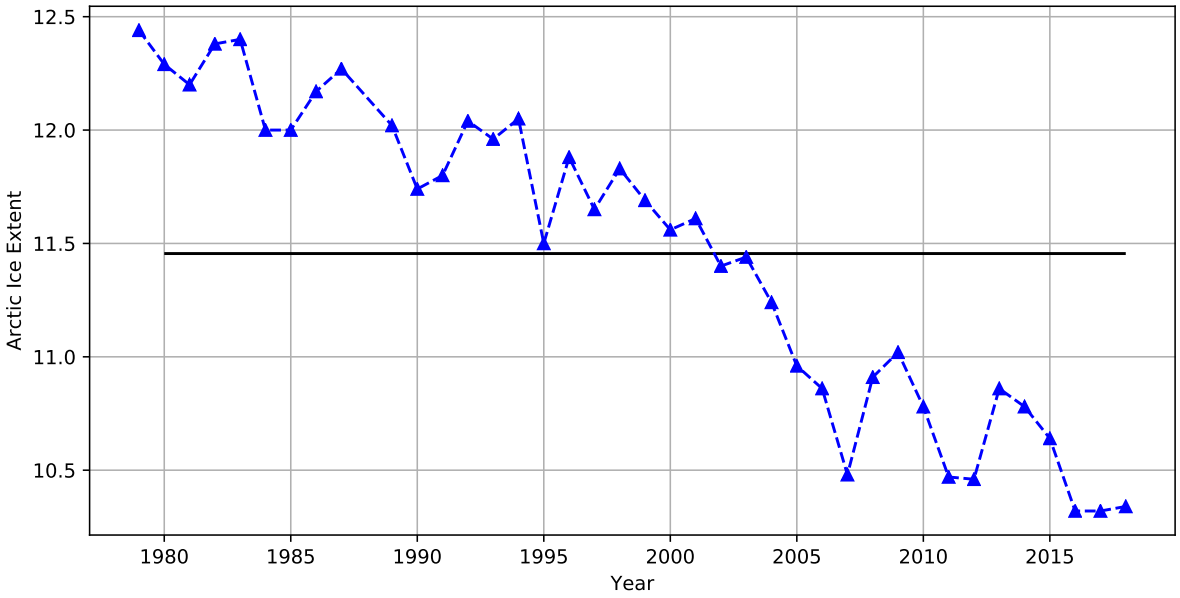

Figure 1.1: Year-averaged Arctic Ice Extent and Mean of the Extents.

Chapter 2

Deterministic Models

2.1 Minimising the Loss for Regression with the Constant Model

To perform SL effectively, we focus on the use of parametric models. In such models, we specify the functions that make predictions from inputs up to a set of parameters, which we will generally denote as θ. By introducing parameters to our model we then have the flexibility to determine our predictions on the basis of experience. In other words, we select the value of our parameters on the based on the dataset we have. This then allows us to realise the general philosophy of ML, such that as the size of the dataset increases and/or the computational time spent examining the dataset increases (i.e. as the experience of our model has grows), our predictions will change (due to the changing choice of parameter values determined by the dataset) in such a way that the performance of our model improves.

For a given $\mathcal{D}$, $L_{\mathcal{D}}$ and parametrised model, everything is then fixed except for the parameters of the model. Therefore, the loss $L_{\mathcal{D}}$ can now be considered as a function of the parameters, $L_{\mathcal{D}}(\theta)$, i.e., for a given input set of parameter values it produces a single value output. Given this fact, it seems natural that we should now fix the value of θ such that the loss-function, $L_{\mathcal{D}}(\theta)$, which measures the error in our predictions, is minimised. In this way, we will select the best model for our task, as measured by the performance measure encoded in the loss.

Before digging into this idea of minimising the loss-function – which is one of the foundational concepts in ML – we will show the process in the simplest possible case: that of single-input, single-output regression using the constant model, i.e.,

$$\hat{y} = \beta_0 \ . \tag{2.1}$$

While such a model might seem absurdly simple, it is actually an important case where the target variable has no dependence on the input. Moreover, the calculation we will perform to minimise the loss-function in this case actually captures a number of important steps that apply in more general cases.

Before evaluating the loss-function, we can rewrite the model in the notation we have developed so far as,

$$\hat{y} = \beta_0 x_0 \ , \tag{2.2}$$

where we have used the definition $x_0 = 1$, for any example i.

This model has only a single parameter, so that $\theta = \{\beta_0\}$. The loss-function is then simply a one-dimensional function (it has only one argument/input), $L_{\mathcal{D}}(\beta_0)$. A typical choice for regression problems is to take the per-example-loss as the squared difference between the predictions and target values. For $K = 1$, this reads;

$$l(\hat{y}, y) = (\hat{y} - y)^2 \, . \tag{2.3}$$

The loss-function – which we will also call the mean-squared-error (MSE) in this case – then takes a

quadratic functional form. For the constant model this is,

$$L_{\mathcal{D}}(\beta_0) = \frac{1}{N}\sum_{i=1}^{N}\left[y^{(i)} - \beta_0\right]^2 \ , \tag{2.4}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left[\beta_0^2 + \left(y^{(i)}\right)^2 - 2y^{(i)}\beta_0\right] \ , \tag{2.5}$$

$$= \beta_0^2 + \bar{S}_{y^2} - 2\bar{S}_y\beta_0 \ . \tag{2.6}$$

The notation $\bar{S}_{...}$ indicates a mean over the variable appearing in the subscript, and helps emphasise in this case that such sums are just fixed coefficients of the loss-function, i.e., they depend only on the chosen examples contained in $\mathcal{D}$ and not the model parameters. Therefore, once $\mathcal{D}$ is chosen the coefficients of the loss-function are all fixed.

All that remains is to find the value of θ^*, defined as the value of the parameter (set) that minimises the loss-function,

$$\theta^* = \arg\min_{\theta} L_{\mathcal{D}}(\theta) \ . \tag{2.7}$$

In general, this is a multidimensional minimisation problem, equivalent to simultaneously minimising every single parameter appearing in θ. In the current case, with just a single parameter, there is just a single minimisation problem to solve,

$$\beta_0^* = \arg\min_{\beta_0} L_{\mathcal{D}}(\beta_0) \ , \tag{2.8}$$

$$= \arg\min_{\beta_0} \left(\beta_0^2 - 2\bar{S}_y\beta_0 - \bar{S}_{y^2}\right) . \tag{2.9}$$

Given that this is a one-dimensional, quadratic loss function, this is essentially the simplest minimisation problem we can encounter. To find the minimum, we begin by differentiating the loss-function with respect to the parameter,

$$\frac{dL}{d\beta_0} = 2\beta_0 - 2\bar{S}_y \ . \tag{2.10}$$

The minimum is then found by solving the equations obtained by setting this derivative to zero,

$$\frac{dL}{d\beta_0} = 0 \ . \tag{2.11}$$

The solution to this is $\beta_0^* = \bar{S}_y$. In words, the optimal choice of the parameter is the mean of the targets in the data set. This completes our model for this problem,

$$\hat{y} = \bar{S}_y x_0 \ . \tag{2.12}$$

Note that this is the constant-model we introduced earlier by hand in the context of the Artic Ice extent. We can now see that this choice can be justified as it is the optimal choice of parametric model where the single target variable is independent of the single input variable.

Example 1: Optimising the Constant Model for Arctic Ice Extent

The constant model for the Arctic Ice extent problem is equivalent to drawing a horizontal line though Fig. 1.1. To find the optimal value of this line, we calculate the loss-function, which is shown in Fig. 2.1. The lowest point, which we can calculate analytically, is located the mean value for the ice extent, 11.46. Therefore, $\beta_0^* = 11.46$ for this model and data set.

2.2 Linear Models

Vector Notation for Linear Models

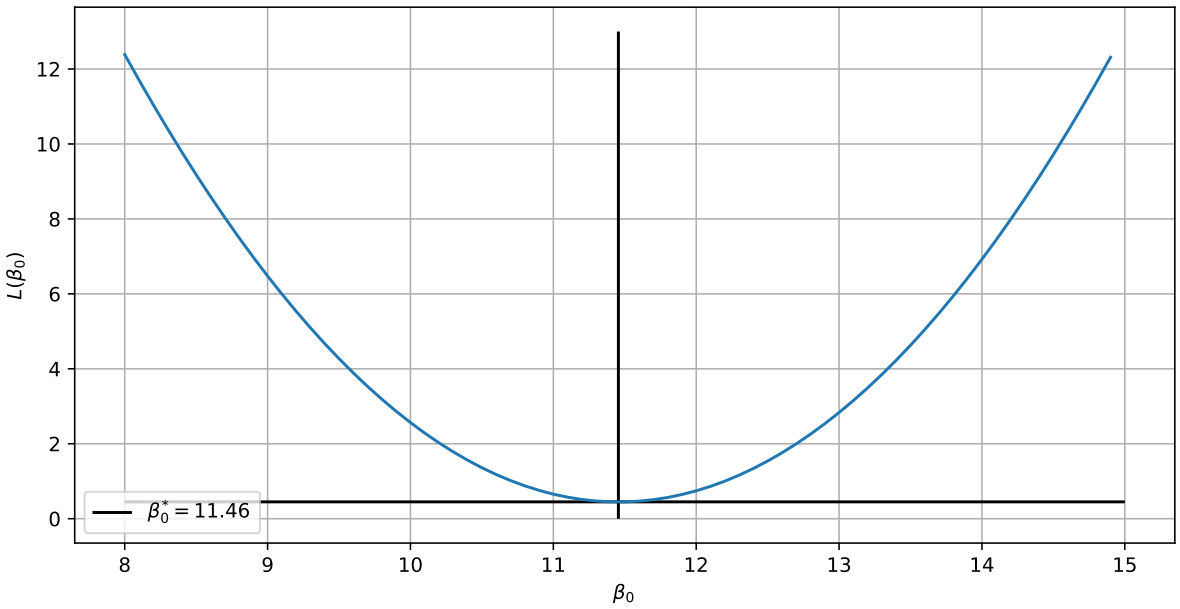

Figure 2.1: Loss-Function for the Arctic Ice with a constant model

So far have only dealt with the explicit minimisation of a loss function with one variable. We will now see how to set up and (in principle) solve the problem with multiple variables, in a way that generalises well to other cases.

The model we will use we will call the linear model. For the case of $K = 1$, on which we will focus, this can be defined as,

$$\hat{y} = \sum_{p=0}^{P} \beta_p x_p \ . \tag{2.13}$$

This is a parametric model with $P + 1$ parameters, $\theta = \{\beta_0, \beta_1, ..., \beta_P\}$. While this model is in many ways the simplest non-trivial one that we can come up with, we will see it is in fact very general, and so can be of practical use in addition to being illustrative.

To make the following calculations simpler (as well as more computationally efficient), we start by making more use of vector notation to replace the explicit sum notation in our expressions for the predictions and the loss-function. For example, for a linear model and assuming $K = 1$ throughout, we can calculate all the predictions for each data point in one go as,

$$\hat{\mathbf{Y}} = \mathbf{X}\boldsymbol{\beta}. \tag{2.14}$$

Here we have introduced the parameter vector, $\boldsymbol{\beta}$, with shape $(P + 1, 1)$ whose components are the parameters β_p, i.e., $\boldsymbol{\beta} = (\beta_0, \beta_1, ..., \beta_P)^T$.

To see why this expression is correct, recall that $\hat{\mathbf{Y}}$, is a vector whose components correspond to the predictions for individual data points. We can now consider the expression (2.14) in components, i.e. consider every row i independently. In that case, we have,

$$\hat{Y}_i = [\mathbf{X}\boldsymbol{\beta}]_i \ , \tag{2.15}$$

$$= \sum_{p=0}^{P} X_{ip}\beta_p \ , \tag{2.16}$$

$$= \sum_{p=0}^{P} x_p^{(i)}\beta_p \ , \tag{2.17}$$

$$= \hat{y}^{(i)} \ , \tag{2.18}$$

$$= \boldsymbol{\beta} \cdot \mathbf{x}^{(i)} = \boldsymbol{\beta}^T \mathbf{x}^{(i)} = (\mathbf{x}^{(i)})^T \boldsymbol{\beta} \ . \tag{2.19}$$

In the last line, we have also expressed the prediction as a dot product between two vectors, which can generally be written in components as,

$$\mathbf{a} \cdot \mathbf{b} = \sum_k a_k b_k \ . \tag{2.20}$$

It is important to note that not only is this notation succinct, but computationally efficient – computations using matrix-vector multiplications tend to have better performance than the simple sum.

Next, we can express the loss-function in a similar manner. As before, we will choose the MSE to be our loss (i.e. the squared difference is the per-example-loss). First, we start by expressing the loss in terms of the target vector, $\mathbf{Y}$, which here is simply a vector with components equal to $y^{(i)}$,

$$\begin{aligned} L_{\mathcal{D}} &= \frac{1}{N}\sum_{i=1}^{N}\left[\hat{y}^{(i)} - y^{(i)}\right]^2 \ , && (2.21)\\ &= \frac{1}{N}\sum_{i=1}^{N}\left[\hat{Y}_i - Y_i\right]^2 \ , && (2.22)\\ &= \frac{1}{N}\sum_{i=1}^{N}\left[\hat{\mathbf{Y}} - \mathbf{Y}\right]_i^2 \ , && (2.23)\\ &= \frac{1}{N}\sum_{i=1}^{N}\left[\hat{\mathbf{Y}} - \mathbf{Y}\right]_i \left[\hat{\mathbf{Y}} - \mathbf{Y}\right]_i \ , && (2.24)\\ &= \frac{1}{N}\left[\hat{\mathbf{Y}} - \mathbf{Y}\right] \cdot \left[\hat{\mathbf{Y}} - \mathbf{Y}\right] \ . && (2.25)\\ & && (2.26) \end{aligned}$$

A vector dotted with itself simply gives the squared-length of that vector, which we will denote using $||\mathbf{v}||_2^2$, for any vector $\mathbf{v}$. Hence, the loss can be written as;

$$\begin{aligned} L_{\mathcal{D}} &= \frac{1}{N}||\hat{\mathbf{Y}} - \mathbf{Y}||_2^2 \ , && (2.27)\\ & && (2.28) \end{aligned}$$

To find an expression for the loss-function, we simply specialise this expression for the loss to our particular parametric model. This is achieved by substituting in the expression for our prediction matrix in terms of the parameters for our chosen model. The loss-function for the linear model is then,

$$L_{\mathcal{D}}(\boldsymbol{\beta}) = \frac{1}{N}||\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}||_2^2 \ . \tag{2.29}$$

This expression tells us that the loss function, $L_{\mathcal{D}}(\boldsymbol{\beta})$, is calculated by 1) applying the data matrix, $\mathbf{X}$, to the parameter vector $\boldsymbol{\beta}$, 2), subtracting this from the target vector $\mathbf{Y}$, 3) calculating the squared-norm of the resulting vector and dividing by N. As $\boldsymbol{\beta}$ changes, i.e. as any of its $P+1$ components change, the value of the loss-function also changes. We are then interested in finding the values of the $P+1$ components of $\boldsymbol{\beta}$ that make L as small as possible.

2.3 Optimisation

Given $L_{\mathcal{D}}(\boldsymbol{\beta})$, we wish to find the vector $\boldsymbol{\beta}^*$ that minimises it. In other words, we want to choose the values of $\beta_0, \beta_1, ..., \beta_P$, so that the value of L is as small as possible. Mathematically, this can be expressed as

$$\begin{aligned} \boldsymbol{\beta}^* &= \text{argmin}_{\boldsymbol{\beta}} L_{\mathcal{D}}(\boldsymbol{\beta}) \ , \\ \beta_0, \beta_1, ..., \beta_P &= \text{argmin}_{\beta_0,\beta_1,...,\beta_P} L_{\mathcal{D}}(\beta_0, \beta_1, ..., \beta_P) \ . \end{aligned} \tag{2.30}$$

Here, the second line, with the components written out explicitly, is there to emphasise that this is general a high dimensional problem – there are $P+1$ parameters, and we must pick their values simultaneously

such that $L_{\mathcal{D}}(\boldsymbol{\beta})$ is minimised. If we imagine that each β_p could take on just two values, the total number of combinations is 2^{P+1}, so picking the single optimal combination is in general exponentially hard. Clearly a brute-force approach of checking every possible combination is not going to work.

Note that, if we are only interested in finding $\boldsymbol{\beta}^*$ that solves this optimisation problem, then we are free to consider any other loss-function, so long as its minimum is in the same location. In particular, we could consider scaling $L_{\mathcal{D}}(\boldsymbol{\beta})$ to create a new $\tilde{L}_{\mathcal{D}}(\boldsymbol{\beta})$, or adding a constant or taking the logarithm. None of these transformations change the location of the minimum (i.e. the components of $\boldsymbol{\beta}^*$ are unchanged). Of course, these transformations do change the value of the loss-function at that minimum and the form of the loss function more generally. However, since we are only interested in the location of the minimum, we have a lot of freedom to make changes to the loss-function that might make computations simpler. Additionally, note we can even change a minimisation problem where the solution is $\boldsymbol{\beta}^*$ into a maximisation problem with the same solution, by applying an overall minus sign to the loss function.

2.4 Basis Expansions

Basis Expansions for Linear Models

Before finding how we can solve the optimisation of (2.30), it is useful to emphasise just how general the linear model is.

The key to seeing this is to realise that "Linear" means linear in the parameters of the model, $\boldsymbol{\beta}$, not that we have to use linear features. For example, for a regression problem with $K = 1$, we might specify our model via a degree P polynomial function,

$$\hat{y} = f(x) = \sum_{p=0}^{P} \beta_p x^p \ . \tag{2.31}$$

In this model, we have only one "basic" input x. However, by taking powers of this we have created other, more complicated (non-linear), features/inputs to work with. These are then combined in a linear way – which is where the parameters of the model enter – to make the predictions. From this perspective we can see it doesn't matter how we come to the features we use, they could be naturally given to us or made by us by combining/transforming others. Ultimately, for a given example/data set all the features are fixed numbers and only the parameters remain to be optimised. Since the parameters appear in a linear way, this is a linear model.

With this view, we can write down an expression for the most general linear model possible for $K = 1$, with $K > 1$ following easily. Starting from an example vector of features $\mathbf{x}$, we process this in P different ways via the functions $h_1(\mathbf{x}), h_2(\mathbf{x}), ..., h_P(\mathbf{x})$, before combining these in a linear manner to give,

$$\hat{y} = \sum_{p=0}^{P} \beta_p h_p(\mathbf{x}) \ , \tag{2.32}$$

where $h_0(\mathbf{x}) = 1$ following the convention introduced previously. Note that the dimension of the original $\mathbf{x}$ need not be P, as is clear from the case of a degree P polynomial function with a single original variable. The above expression is sometimes called a basis expansion since, as in the case of polynomials, we could in principle describe any function $f(x)$ we like this way by selecting a set of functions which are complete (form a basis).

To make the fact this is indeed a linear model explicit, we can also write the above expression as,

$$\hat{y} = \boldsymbol{\beta}^T \tilde{\mathbf{x}} \ , \tag{2.33}$$

where $\tilde{\mathbf{x}}$ is the $P + 1$ dimensional vector of constructed features, i.e. it has components

$$\tilde{\mathbf{x}}_p = h_p(\mathbf{x}) \ . \tag{2.34}$$

Written in terms of $\tilde{\mathbf{x}}$, it is clear that the functional form of $\hat{y}$ with respect to the parameters is identical to that of the case we have looked at previously. Therefore, all the results we have derived so far can be applied directly, the only difference is that we make use of the constructed features $\tilde{\mathbf{x}}$ rather than

the original inputs $\mathbf{x}$, i.e. everywhere we replace $\mathbf{x} \to \tilde{\mathbf{x}}$. In fact, since ultimately there is no difference between the features we are "given" as inputs and those we construct, we will drop the tilde notation altogether, assuming that the dataset we are working with, $\mathcal{D}$, has already been processed to our liking into P input features per example.

Example 1: Basis Expansions for Arctic Ice

In a linear model with quadratic basis expansion, $P = 2$, we take the original feature, the year, and process it via;

$$h_1(x) = x \tag{2.35}$$
$$h_2(x) = x^2 \ , \tag{2.36}$$

such that $P = 2$.

The dataset we will now work with is encoded in the design matrix, $\mathbf{X}$, which now has shape $(N, 3)$;

$$\mathbf{X} = \begin{pmatrix} 1 & 1979 & 1979^2 \\ 1 & 1980 & 1980^2 \\ ... & ... & ... \\ 1 & 2018 & 2018^2 \end{pmatrix} \ , \tag{2.37}$$
$$= \begin{pmatrix} 1 & 1979 & 3916441 \\ 1 & 1980 & 3920400 \\ ... & ... & ... \\ 1 & 2018 & 4072324 \end{pmatrix} \ . \tag{2.38}$$

The target matrix, $\mathbf{Y}$, is unchanged.

Note that we could also consider different basis expansions using any functions we like. For example,

$$h_1(x) = \log(x) \tag{2.39}$$
$$h_2(x) = \sqrt{x} \ , \tag{2.40}$$
$$h_3(x) = e^x \ , \tag{2.41}$$

would create a data set with $P = 3$ features to work with.

2.5 Optimisation for the Linear Model

Normal Equations for the Linear Model

We now turn to the problem of minimising the MSE loss-function for the linear model in general;

$$\boldsymbol{\beta}^* = \text{argmin}_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) \ .$$

In this case we have $P + 1$ parameters, $\beta_0, ..., \beta_P$, acting as inputs for the function L.

The minimum of a multi-variable function can found by setting all the partial derivatives of the function to zero simultaneously. For the linear model and MSE loss, we have $P + 1$ partial derivatives we can compute. Just as it is convenient to collect together the parameters together as components of a single vector $\boldsymbol{\beta}$, it is helpful to collect the partial derivatives together as the components of another vector, known as the gradient, $\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta})$. The components of this vector are;

$$\left[\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta})\right]_p = \frac{\partial L(\boldsymbol{\beta})}{\partial \beta_p} \ , \tag{2.42}$$

i.e.

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial L(\boldsymbol{\beta})}{\partial \beta_0} \\ \frac{\partial L(\boldsymbol{\beta})}{\partial \beta_1} \\ ... \\ \frac{\partial L(\boldsymbol{\beta})}{\partial \beta_P} \end{pmatrix} . \tag{2.43}$$

Since at the minimum (or any stationary point) all partial derivatives are simultaneously zero, we can write the condition for $\boldsymbol{\beta}^*$ succinctly as

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = \mathbf{0} \ , \tag{2.44}$$

where

$$\mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ ... \\ 0 \end{pmatrix} \ . \tag{2.45}$$

This vector-equation is often called the normal equation.

To evaluate the gradient of the linear model's loss function, it is clearest in the first instance to proceed component-wise, i.e. for each β_p separately, using the sum notation. First, recalling that only $\hat{y} = \boldsymbol{\beta}^T \mathbf{x}$ is a function of $\boldsymbol{\beta}$, we apply the chain rule;

$$\frac{\partial L}{\partial \beta_p} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \left[(y^{(i)} - \hat{y}^{(i)})^2 \right]}{\partial \beta_p} \ , \tag{2.46}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \left[(y^{(i)} - \boldsymbol{\beta}^T \mathbf{x}^{(i)})^2 \right]}{\partial \beta_p} \ , \tag{2.47}$$

$$= \frac{1}{N} \sum_{i=1}^{N} 2(y^{(i)} - \boldsymbol{\beta}^T \mathbf{x}^{(i)}) \frac{\partial (y^{(i)} - \boldsymbol{\beta}^T \mathbf{x}^{(i)})}{\partial \beta_p} \ , \tag{2.48}$$

$$= \frac{2}{N} \sum_{i=1}^{N} (\boldsymbol{\beta}^T \mathbf{x}^{(i)} - y^{(i)}) \frac{\partial \boldsymbol{\beta}^T \mathbf{x}^{(i)}}{\partial \beta_p} \ . \tag{2.49}$$

The second factor is simple to evaluate;

$$\frac{\partial \boldsymbol{\beta}^T \mathbf{x}^{(i)}}{\partial \beta_p} = \frac{\partial \left(\beta_0 x_0^{(i)} + \beta_1 x_1^{(i)} + ... + \beta_P x_P^{(i)} \right)}{\partial \beta_p} \ , \tag{2.50}$$

$$= x_p^{(i)} \ . \tag{2.51}$$

Substituting this in we have;

$$\frac{\partial L}{\partial \beta_p} = \frac{2}{N} \sum_{i=1}^{N} x_p^{(i)} (\boldsymbol{\beta}^T \mathbf{x}^{(i)} - y^{(i)}) \tag{2.52}$$

$$\tag{2.53}$$

Therefore, the gradient as a whole can be expressed as;

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = \frac{2}{N} \begin{pmatrix} x_0^{(i)} (\boldsymbol{\beta}^T \mathbf{x}^{(i)} - y^{(i)}) \\ x_1^{(i)} (\boldsymbol{\beta}^T \mathbf{x}^{(i)} - y^{(i)}) \\ ... \\ x_P^{(i)} (\boldsymbol{\beta}^T \mathbf{x}^{(i)} - y^{(i)}) \end{pmatrix} \tag{2.54}$$

To express this in terms of the design and target matrices, we use the definitions of their components;

$$\frac{\partial L}{\partial \beta_p} = \frac{2}{N} \sum_{i=1}^{N} x_p^{(i)} ([\mathbf{X}\boldsymbol{\beta}]_i - Y_i) \tag{2.55}$$

$$= \frac{2}{N} \sum_{i=1}^{N} X_{ip} \left[\mathbf{X}\boldsymbol{\beta} - \mathbf{Y} \right]_i \tag{2.56}$$

$$= \frac{2}{N} \sum_{i=1}^{N} \left[\mathbf{X}^T \right]_{pi} \left[\mathbf{X}\boldsymbol{\beta} - \mathbf{Y} \right]_i \tag{2.57}$$

$$= \frac{2}{N} \left[\mathbf{X}^T (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) \right]_p \ . \tag{2.58}$$

Therefore,

$$\nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = \frac{2}{N}\mathbf{X}^T \left[\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}\right] \ . \tag{2.59}$$

The normal equations can then be expressed as a single vector equation;

$$\frac{2}{N}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) = \mathbf{0} \ , \tag{2.60}$$

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{Y} = \mathbf{0} \ , \tag{2.61}$$

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{Y} \ . \tag{2.62}$$

This can be solved by using a matrix inverse, which finally determines the optimal parameter vector for the linear model:

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \ . \tag{2.63}$$

Example: Linear Models for the Arctic Ice Extent

Combining basis expansions with the general solution to the linear model, we can easily study models for the Arctic ice regression problem, where the predicted extents are non-linear in the input year.

Three example models, using polynomial basis expansions, are shown in Fig. 2.2. Note again that the models produce curved lines of predictions, even though the model itself is a Linear model, since Linear in this context means linear in the parameters. This is important to realise, because we can then see that we could, in principle, construct Linear models of very high complexity by considering very large basis expansions, e.g. very high order polynomials. Unsurprisingly, such models will be able to fit the data we provide better than those with lower complexity. In fact, the performance of our models seems to increase without end in this sense. Ultimately we could choose a polynomial basis expansion of such high degree that we fit our data set perfectly. Yet, we know from experience that such models tend to be quite poor at making predictions when challenged by examples from outside the experience. All of this is related to another key consideration of machine learning, that of generalisation and regularisation, which we will examine more fully later on.

2.6 Linear Models with $K > 0$.

Recall that when we have multiple target variables to predict, i.e. we have a vector of target variables for each input, we can encode the targets in a matrix $\mathbf{Y}$ of shape (N, K). As such, our model's predictions must should be contained in a matrix $\hat{\mathbf{Y}}$ of shape (N, K). The rows of this matrix contain our model's predictions on each example $i = 1, 2, ..., N$ in $\mathcal{D}$. These predictions are denoted $\hat{\mathbf{y}}^{(i)}$. Note that when writing the predictions as vectors, the shape of $\hat{\mathbf{y}}^{(i)}$ is $(K, 1)$. Therefore, to construct $\hat{\mathbf{Y}}$ these are first transposed to get $\left(\hat{\mathbf{y}}^{(i)}\right)^T$ which has shape $(1, K)$ (i.e. one row and K columns). These N vectors are then stacked together to form $\hat{\mathbf{Y}}$.

For the linear model, the prediction matrix can be written as,

$$\hat{\mathbf{Y}} = \mathbf{X}\boldsymbol{\beta} \ . \tag{2.64}$$

In this equation, $\boldsymbol{\beta}$ is now a matrix of parameters with shape $(P+1, K)$. This can be seen by considering the shapes on the right hand-side: $\mathbf{X}$ has shape $(N, P+1)$ and therefore must be multiplied by a matrix of shape $(P+1, K)$ to produce a matrix of shape (N, K) and thus agree with the shape of $\hat{\mathbf{Y}}$ on the left-hand side.

To see why (2.64) is indeed the linear model as desired, we consider each of the $i = 1, 2, ..., N$ examples

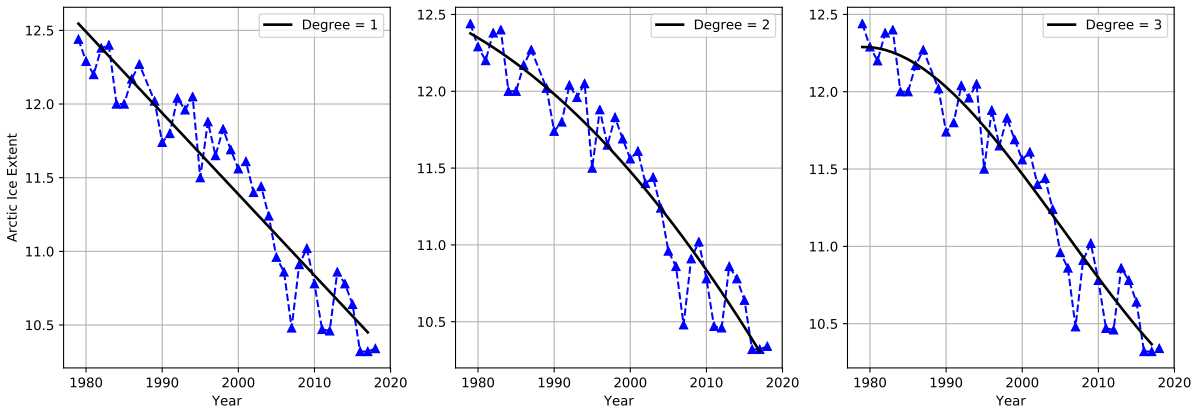

Figure 2.2: Linear Models with Polynomial Basis Expansion for Year-averaged Arctic Ice Extent

independently. That is, we consider the components of the equation;

$$[\hat{\mathbf{Y}}]_{ik} = [\mathbf{X}\boldsymbol{\beta}]_{ik} \ , \tag{2.65}$$

$$= \sum_{p=0}^{P+1} [\mathbf{X}]_{ip}[\boldsymbol{\beta}]_{pk} \ , \tag{2.66}$$

$$\hat{y}_k^{(i)} = \sum_{p=0}^{P+1} x_p^{(i)}[\boldsymbol{\beta}]_{pk} \ . \tag{2.67}$$

$$\tag{2.68}$$

If we collect all the $P+1$ input variables into a single vector as usual, then this can also be written as,

$$\hat{y}_k^{(i)} = \boldsymbol{\beta}_k^T \mathbf{x}^{(i)} \ . \tag{2.69}$$

Note that in this final expression, $\boldsymbol{\beta}_k$ is a vector, with shape $(P+1, 1)$ just as for the linear model before. However, we now have K such vectors indexed by the subscript $k = 1, 2, .., K$. Therefore, in the linear model with $K > 1$ we effectively have an independent vector of parameters for every target variable k. To express this in one go, we stack these parameter vectors into a single parameter matrix. This, in general, to optimise the linear model we must find the optimal values of the $K \times (P+1)$ components of this matrix.

Chapter 3

Probabilistic Models

Previously, we have considered the basic components of machine learning algorithms: Tasks (Ts), performance measures (PMs) and experience (E). The example T that we have focussed on is that of regression, phrased as the prediction of a numerical quantity given some input.

In cases where we are interested in quantifying the uncertainty in our predictions, we would like to predict not just a single value for some quantity given an input, but the probability of that quantity taking on some chosen value. In other words, for a given example input $\mathbf{x}$, we do not want to simply return a predicted value, $\hat{y} = f(\mathbf{x})$, but a probability (for discrete target variables) or a probability density (for continuous target variables), $p\left(y|\mathbf{x}\right)$, which is a function of both the input value, $\mathbf{x}$, and the output value, y (assuming $K = 1$ for now). Of course, this function should be chosen so that we can interpret it appropriately as a probability or a probability density for continuous variables.

Note that the probabilistic way of making predictions is more general than the previous "deterministic" one, where we return a single value. In particular, for discrete target variables we could always choose a function such that our probability is one for only a single value of $y = f(\mathbf{x})$, i.e. $p(y = f(\mathbf{x})|\mathbf{x}) = 1$. For a continuous variable, the same deterministic model can be achieved with a delta-function probability density, which is peaked around $y = f(\mathbf{x})$.

Thinking in terms of probabilities also helps clarify what we mean by supervised and unsupervised learning. In both cases, we can consider having a data set consisting of examples of some set of variables, $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, ..., \mathbf{z}^{(N)}$.

In the supervised learning case, we are particularly interested in one of these variables, which we call the target, denoted y, and we split up each example into a pair as $\mathbf{z} = (\mathbf{x}, y)$. Supervised learning tasks consist of constructing a model for the conditional probability of y, given $\mathbf{x}$, i.e. $p\left(y|\mathbf{x}\right)$.

In the unsupervised case, we do not have a special interest in a particular variable, so we do not split up our $\mathbf{z}$s, and can simply write $\mathbf{z} = \mathbf{x}$. In this case, we are then interested in simply modelling $p\left(\mathbf{x}\right)$ directly.

While the probabilistic view helps us separate supervised learning from unsupervised learning, it also shows that the difference is not clean cut. This is because we can always convert between joint probability distributions and conditional probabilities via,

$$p\left(y, \mathbf{x}\right) = p\left(y|\mathbf{x}\right) p\left(\mathbf{x}\right) \ . \tag{3.1}$$

Therefore, in a supervised learning problem, it is possible to build a model of $p\left(y|\mathbf{x}\right)$ by considering an unsupervised learning problem of modelling $p\left(y, \mathbf{x}\right)$ (the other factor $p\left(\mathbf{x}\right) = \sum_{y'} p\left(y', \mathbf{x}\right)$ just provides normalisation). Conversely, an unsupervised learning problem of modelling $p(\mathbf{x})$ can be achieved by decomposing a joint probability distribution (via the chain-rule which generalises the above equation) and modelling the resulting conditional probabilities. For example, if $\mathbf{x} = (x_1, x_2, x_3)$, then we could write;

$$p(\mathbf{x}) = p(x_1, x_2, x_3) \tag{3.2}$$

$$= p(x_1|x_2, x_3)p(x_2|x_3)p(x_3) \ . \tag{3.3}$$

We will focus on modelling the conditional probability distribution $p(y|\mathbf{x})$ directly, as is usual with supervised learning, starting with the case of linear regression.

3.1 Probabilistic Linear Models for Regression

Gaussian Linear Model

A natural way to extend the (deterministic) linear model is to take the model we had previously,

$$\hat{y} = \boldsymbol{\beta}^T \mathbf{x}, \tag{3.4}$$

and consider a function for the probability density so that both the expected value and most probable value of y is given by $\hat{y}$. Most commonly, this is achieved using a Gaussian function with mean at $\hat{y}$;

$$p(y|\mathbf{x}) = \mathcal{N}\left(y; \hat{y} = \boldsymbol{\beta}^T \mathbf{x}, \sigma^2\right) , \tag{3.5}$$

$$= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-1}{2\sigma^2}\left(y - \boldsymbol{\beta}^T \mathbf{x}\right)^2\right] . \tag{3.6}$$

Note that in this probabilistic model we have now an additional parameter, σ^2, which controls the width of the Gaussian and is equal to the variance of the distribution of predictions for the target variable. Therefore, in total we have $P+2$ parameters, $\theta = \{\boldsymbol{\beta}, \sigma^2\} = \{\beta_0, \beta_1, ..., \beta_P, \sigma^2\}$. If we consider removing this parameter by taking $\sigma^2 \to 0$, then our probabilities become more and more peaked around $\hat{y}$, thus reproducing our previous deterministic model.

While we will focus on the Gaussian Linear Model for regression, note that there are two potential changes we could make to consider a wider variety of other models. Firstly, we could retain the Gaussian part of the structure and write more generally;

$$p(y|\mathbf{x}) = \mathcal{N}\left(y; \mu(\mathbf{x}), \sigma^2\right) , \tag{3.7}$$

so that the mean of the Gaussian is a function of the input variable. By choosing the functional form of $\mu(\mathbf{x})$ we can come to a variety of Gaussian models for regression tasks. The Gaussian Linear Model is then formed by taking the special case of a Gaussian model with $\mu(\mathbf{x}) = \boldsymbol{\beta} \cdot \mathbf{x}$.

Secondly, while in many situations a Gaussian model is a good choice, with a balance between simplicity and accuracy, in some scenarios such as for data sets with significant outliers, other distributions with heavier tails maybe more appropriate. For instance, one can replace the Gaussian distribution with the Laplace distribution in this case, though the resulting mathematics becomes more sophisticated. This offers another potential avenue for developing machine learning models for regression.

Gaussian Linear Model for Arctic Ice Extent

Previously, we have looked at the Arctic Ice problem using a data set with only one value of the target variable per year. To increase the complexity of this problem we can now consider a data set where the Arctic Ice extent has been measured every month of the year. Therefore, our data set will now contain twelve different values for the Arctic ice for each year.

While in principle we could label each variable separately by the month if we knew which measurement corresponded to which month, we will instead assume that only the year of the measurement is known. As such, it makes sense that we should now model the conditional probability of the ice extent given a particular year, as there is naturally a distribution of values in the extent that we would like to describe.

In this data set, $N = 40 \times 12 = 468$ (for 12 measurements per year between 1979 and 2018). The first and last few points in the data set are;

$$\mathcal{D} = \{(1979, 15.41), (1979, 16.18), (1979, 15.45), ..., (2018, 9.49), (2018, 11.74)\} . \tag{3.8}$$

Note the fact that each year is repeated 12 times.

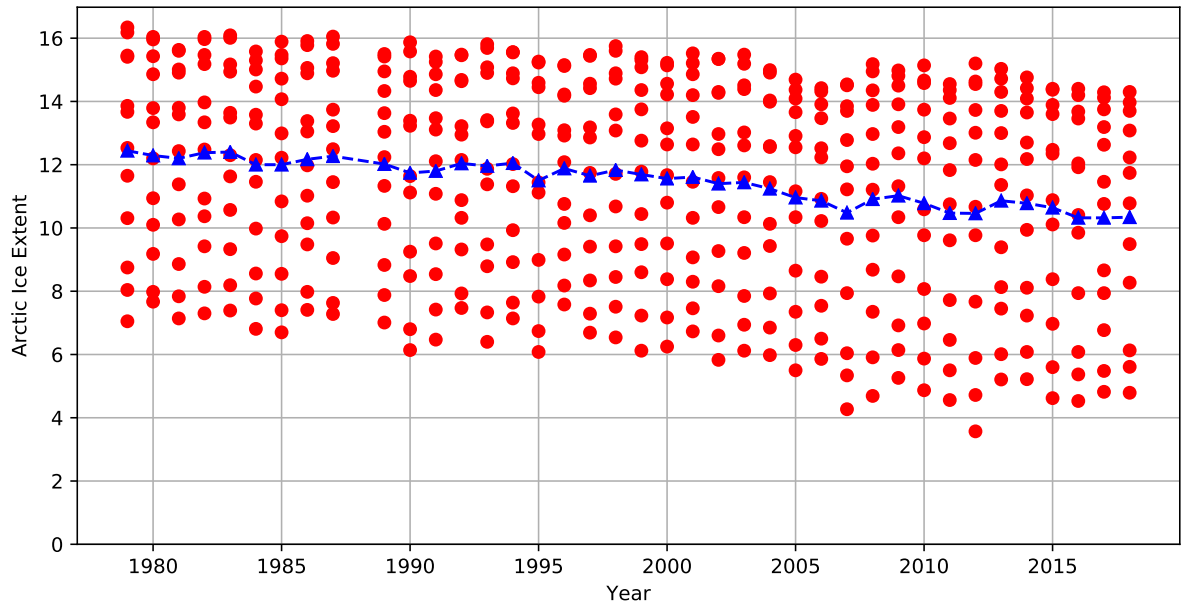

Figure 3.1: Monthly measurements of the Arctic Ice Extent plotted per year (red circles), along with the mean (blue-triangles), which we studied previously.

The design matrix for this data set is;

$$\mathbf{X} = \begin{pmatrix} 1 & 1979 \\ 1 & 1979 \\ 1 & 1979 \\ \dots & \dots \\ 1 & 2018 \\ 1 & 2018 \end{pmatrix} . \tag{3.9}$$

The target vector is;

$$\mathbf{Y} = \begin{pmatrix} 15.41 \\ 16.18 \\ 15.45 \\ \dots \\ 9.49 \\ 11.74 \end{pmatrix} . \tag{3.10}$$

The data set is shown in Fig. 3.1.

If we assume a linear model with linear basis expansion (i.e. a degree-one polynomial), then the basic $\mathbf{X}$ above is also the design matrix we will use for our model. The parameter vector in this case is simply,

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} . \tag{3.11}$$

Our predictions for this model are now probabilities of finding a certain ice extent in a given year. For example, in the year 2018, the predicted probabilities are;

$$p(y|\mathbf{x} = (1, 2018)^T) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-1}{2\sigma^2}(y - \beta_0 - 2018\beta_1)^2\right] . \tag{3.12}$$

Of course, to actually calculate these probabilities we need to choose values of all the parameters for the model, $\theta = (\sigma^2, \beta_0, \beta_1)$. How to do this in the case of a probabilistic model is, of course, a key question. One option – which turns out to be very reasonable – is to pick $\boldsymbol{\beta}$ in the same manner as for

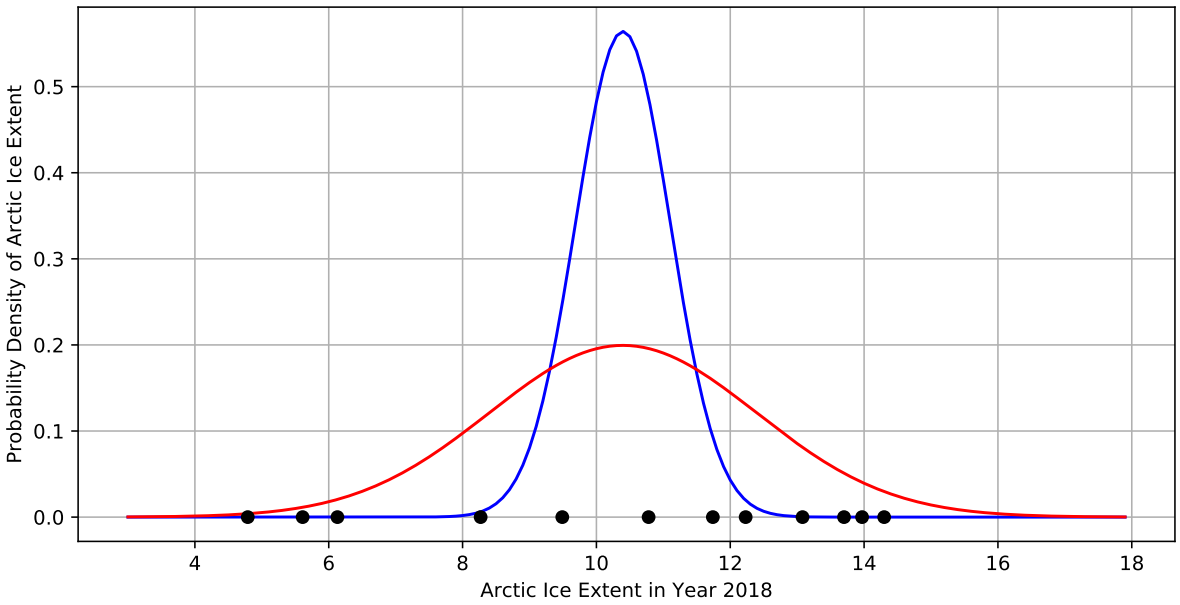

Figure 3.2: Probability Density for Gaussian Linear Models with $\sigma^2 = 0.5$ (blue-line) and $\sigma^2 = 4$ (red-line) of Arctic Ice Extent in Year 2018. The twelve measured data points for this year are indicated by black circles.

the deterministic models (i.e. minimise the mean-squared-error over the data set). However, this still leaves us with the extra parameter, σ^2, that does not appear in the deterministic case.

As an example, we pick the values of $\beta_0, \beta_1 = 122, -0.0551$, calculated using the normal equations found previously, and two values of $\sigma^2 = 0.5, 4$. The probability density of various ice extents are shown in Fig. 3.2, along with the data points for 2018 year for comparison.

3.2 Maximum Likelihood Estimation

Negative Log-Likelihood

Now that we have chosen our task and defined our model, we need to decide what our performance measure should be when considering a probabilistic model. For our deterministic model, in the regression case, we had been using the MSE. In fact, for Gaussian linear models, this corresponds to a probabilistic performance measure known as the likelihood, in the sense that the same value of $\boldsymbol{\beta}^*$ will provide the best performance according to both measures. As the use of the likelihood is somewhat more general, this helps to justify the use of the MSE previously.

Let us consider a probabilistic model for our supervised learning problem, $p\left(y|\mathbf{x};\theta\right)$, where we will keep the dependence of the model on its set of parameters explicit for now. When we use the likelihood as our performance measure, we apply the Maximum Likelihood Principle to determine our parameters (a process which is known as Maximum Likelihood Estimation (MLE)).

In MLE, we select the best-value of θ^* by maximising the likelihood, $\mathcal{L}$, of our data, which is defined as the probability our model predicts of the observing the data set. If we make the assumption that our data is produced in an independent, identically distributed (iid) fashion, then the probability of the observing $\mathcal{D}$ is simply the product of observing each data point individually:

$$\mathcal{L} = \prod_{i=1}^{N} p(y^{(i)}|\mathbf{x}^{(i)};\theta) \ . \tag{3.13}$$

Maximum likelihood estimation then suggests that we should select our parameters via;

$$\theta^* = \text{argmax}_\theta \prod_{i=1}^{N} p(y^{(i)}|\mathbf{x}^{(i)};\theta) \ . \tag{3.14}$$

The basic intuition behind the above maximisation is that, because $\mathcal{D}$ contains the actual, real observations, a correct model would predict those observations as the most probable things to see. Later, we will make this idea more precise when we consider Bayesian probabilities.

In practice, we often like to deal with minimisation problems involving sums, rather than maximisations involving products. Indeed, this is what we achieved when defining a loss, L, for our deterministic models. Here, we will now start from MLE to again form a loss L (i.e. a function of the predictions and targets) that for parametric models can be minimised as a loss-function, $L(\theta)$, to determine θ^*.

To begin, we can re-express the problem of MLE as that of negative-log-likelihood (NLL) minimisation. The negative log-likelihood is defined as minus the log of the likelihood;

$$\begin{aligned} \text{NLL} &= -\frac{1}{N}\log\left[\mathcal{L}\right] && (3.15)\\ &= -\frac{1}{N}\log\left[\prod_{i=1}^{N} p(y^{(i)}|\mathbf{x}^{(i)};\theta)\right] , && (3.16)\\ &= -\frac{1}{N}\sum_{i=1}^{N}\log\left[p(y^{(i)}|\mathbf{x}^{(i)};\theta)\right] , && (3.17) \end{aligned}$$

where we have also divided by a factor of N, which is convenient for consistency with previous notation.

Since neither taking the logarithm of a function nor scaling it changes the location of its maxima/minima, changing $\mathcal{L}$ to the NLL in this way will not change θ^*. The only difference will come from the overall minus sign, which will change the maximum of $\mathcal{L}$ into a minimum of NLL. Therefore, MLE can be equivalently re-expressed as the minimisation of the NLL;

$$\begin{aligned} \theta^* &= \text{argmax}_\theta \mathcal{L} , && (3.18)\\ &= \text{argmin}_\theta \text{NLL} && (3.19)\\ &= \text{argmin}_\theta \left(-\frac{1}{N}\sum_{i=1}^{N}\log\left[p(y^{(i)}|\mathbf{x}^{(i)};\theta)\right]\right) . && (3.20) \end{aligned}$$

This is exactly the form we want of our loss L. Therefore, for probabilistic models, we will consider the NLL as our loss. Note that, as for the MSE, the definition of L is independent of the specific model itself and depends only on the data and the predictions, not specifically how we make those predictions. This will be determined by the model we choose, and lead to the definition of a Loss-function.

In summary, we will focus on the minimisation of the NLL as the way to choose the best parameters for our probabilistic models. Therefore, once the model and data are fixed, the NLL is the loss-function for such models.

Minimisation of NLL for the Gaussian Linear Model

Let us apply MLE to the Gaussian linear model, $p\left(y|\mathbf{x};\theta\right) = p\left(y|\mathbf{x};\boldsymbol{\beta},\sigma^2\right)$. An expression for the NLL,

which forms our loss-function, $L(\theta)$, for this problem can be calculated as;

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[\log p(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\beta},\sigma^2)\right] , \tag{3.21}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(\frac{-1}{2\sigma^2}\left(y^{(i)}-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)^2\right)\right] , \tag{3.22}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}\left[\log\frac{1}{\sqrt{2\pi\sigma^2}}+\log\exp\frac{-1}{2\sigma^2}\left(y^{(i)}-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)^2\right] , \tag{3.23}$$

$$= -\log\frac{1}{\sqrt{2\pi\sigma^2}}-\frac{1}{N}\sum_{i=1}^{N}\frac{-1}{2\sigma^2}\left(y^{(i)}-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)^2 , \tag{3.24}$$

$$= \frac{1}{2}\log 2\pi+\frac{1}{2}\log\sigma^2+\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y^{(i)}-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)^2 , \tag{3.25}$$

$$= \frac{1}{2}\log 2\pi+\frac{1}{2}\log\sigma^2+\frac{1}{2\sigma^2}\mathrm{MSE}(\boldsymbol{\beta}) . \tag{3.26}$$

To find the minimum of this loss function, we can then take partial derivatives with respect to all the parameters;

$$\frac{\partial L}{\partial\beta_p} = \frac{1}{2\sigma^2}\frac{\partial\mathrm{MSE}(\boldsymbol{\beta})}{\partial\beta_p} , \tag{3.27}$$

$$\frac{\partial L}{\partial\sigma^2} = \frac{1}{2\sigma^2}-\frac{1}{2\sigma^4}\mathrm{MSE}(\boldsymbol{\beta}) , \tag{3.28}$$

and then set them equal to zero. The normal equations are therefore,

$$\frac{\partial\mathrm{MSE}(\beta)}{\partial\boldsymbol{\beta}_p} = 0 , \tag{3.29}$$

$$\mathrm{MSE}(\boldsymbol{\beta}) = \sigma^2 . \tag{3.30}$$

The first set of these equations (for $p = 0, 1, ..., P$) are just the normal equations for the (deterministic) linear model found previously. Therefore, the value of $\boldsymbol{\beta}^*$ is the same for MLE as we found by minimising the MSE. This helps to justify our use of the MSE. Of course, in addition to the $\boldsymbol{\beta}$ parameters of our deterministic model, we now also have another parameter σ^2 which appears only in our probabilistic model. Simply substituting the value of $\boldsymbol{\beta}^*$ that we would find from the first set of equations into the second gives,

$$(\sigma^2)^* = \mathrm{MSE}(\boldsymbol{\beta}^*) . \tag{3.31}$$

Recalling that this parameter determines the width of the Gaussian we used, then it also determines the predicted spread in data. The MSE is essentially a measure of this spread away from the mean (given by $\boldsymbol{\beta}^T\mathbf{x}$) and so it makes sense that, in the second equation, MLE has determined the value of σ^2 to be equal to the MSE, evaluated at $\boldsymbol{\beta}^*$.

MLE for the Gaussian Linear Model of Arctic Ice Extent

To calculate θ^* for a Gaussian linear model we can first calculate $\boldsymbol{\beta}^*$ using the expression we calculated before from the MSE:

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} . \tag{3.32}$$

This gives, assuming a degree-one polynomial model, $\boldsymbol{\beta}^* = (\beta_0^*, \beta_1^*) = (122, -0.0551)$. Substituting these values in the MSE allows us to calculate the remaining parameter, σ^2,

$$(\sigma^2)^* = \frac{1}{N}||\mathbf{Y}-\mathbf{X}\boldsymbol{\beta}^*||_2^2 , \tag{3.33}$$

$$= 9.989 , \tag{3.34}$$

so that $\sigma^* = 3.16$.

The resulting model is illustrated in Fig. 3.3.

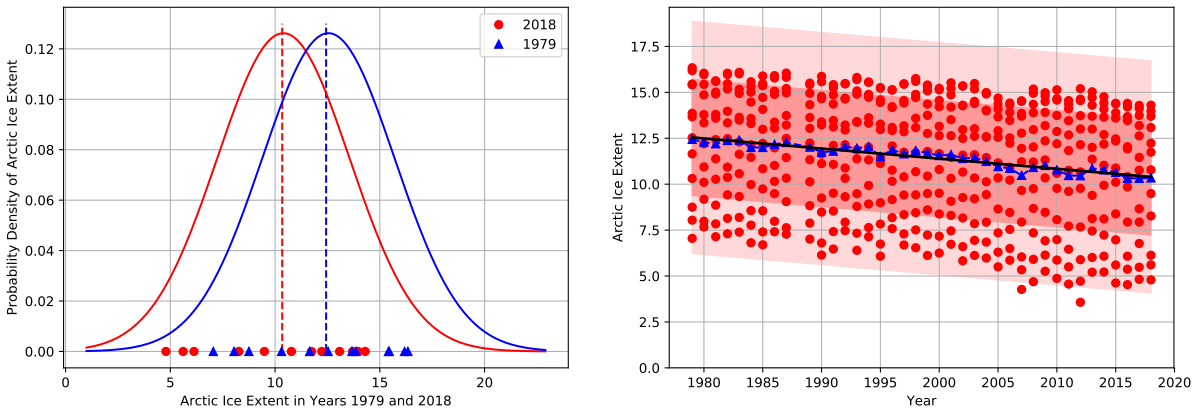

Figure 3.3: Gaussian Linear Model for Arctic Ice Extent, fit using maximum likelihood estimation. The left-hand plot shows the predicted probability density for 1979 and 2018 (blue and red respectively). The width of both Gaussians are equal by assumption in the model, while the location of the means differ. The 12 data points for each year are shown as red circles/blue triangles for comparison. The right-hand plot shows the all the data, plotted as red circles, along with the annual means of the data, shown as blue triangles. This is compared to the predicted mean of the Gaussian linear model, shown as a solid blue line. The width of the Gaussian is indicated by the shaded regions; the darker red shaded region represents one σ^* from the mean, while the lighter region represented two σ^*.

3.3 Probabilistic Models for Classification

Encodings for Categorical Variables

In addition to regression tasks, another common, broad type of task is classification. In such tasks, one considers an input example, $\mathbf{x}$, and predicts the class to which that example belongs.

As an example, we will consider predicting the level of damage to vegetation surrounding a volcano, given a concentration of SO_2. The vegetation damage is described by a vegetation index with three categories – healthy, damaged or devastated. Therefore, unlike the other quantities we have looked at so far, the vegetation index is a *categorical variable*. As such, this task is a classification task; we must assign the correct category to the vegetation given a value of SO_2 concentration (which is a continuous numerical variable).

Before looking at how to construct machine learning models for classification, we must first carefully consider the nature of categorical variables. First, it is important to note that in categorical variables there is no intrinsic ordering between the classes, which sets such variables apart from discrete numerical quantities. Yet, this presents a problem; to make calculations and develop our models we typically like to deal with numerical quantities only. Therefore, we must find an appropriate *encoding* for the categorical variables. By appropriate, we mean an encoding that captures the meaning of the classes, without imposing too much unnecessary extra structure.

The simplest example of an encoding for categorical variables is a discrete variable labelling. In our example, healthy, damaged and devastated might become $1, 2$ and 3 respectively. From the perspective of a simple labelling, this encoding is sensible. However, note that when we come to produce algorithms, this encoding can be misleading. For instance, we can obviously perform any arithmetic operations we like on the labels (e.g. $1 + 2 = 3$) but this does not have any direct meaning in terms of the categories. This is what is meant by the encoding having "extra structure" beyond that present in the categories. Nonetheless, in the case of vegetation damage, this $1, 2, 3$ labelling seems reasonable as in some sense, healthy $<$ damaged $<$ devastated, just as $1 < 2 < 3$, though of course "$<$" does not have any quantitative meaning for the categories, so we should be careful.

Typically, a better encoding for categorical variables can be achieved using vectors (this is sometimes called *one-hot encoding*). In this encoding, we associate each category to a basis vector (a set of basis vectors, $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, ...$, are vectors that all have length one; $\boldsymbol{i} \cdot \boldsymbol{i} = \boldsymbol{j} \cdot \boldsymbol{j} = \boldsymbol{k} \cdot \boldsymbol{k} = ... = 1$ and are mutually orthogonal, i.e. their dot products with respect to each other are zero, e.g., $\boldsymbol{i} \cdot \boldsymbol{j} = 0$).

For the three categories in the vegetation index, we thus have three basis vectors;

$$\text{healthy} \to \begin{pmatrix}1\\0\\0\end{pmatrix}, \text{ damaged} \to \begin{pmatrix}0\\1\\0\end{pmatrix}, \text{ devastated} \to \begin{pmatrix}0\\0\\1\end{pmatrix} . \tag{3.35}$$

The advantage of this encoding is that the categories have been encoded in a set of *mutually orthonormal* vectors. Therefore, from the perspective of the vector space they inhabit, they are all equal – they have equal length and are all equally far from each other (at right angles). This means none of the categories is picked out as special by the encoding, which could introduce biases in our models that are not present in the original task formulation.

The data set we will use for this example consists of $N = 61$ points. The first few data points are:

$$\mathcal{D} = \{(22.01, \text{healthy}), (44.28, \text{healthy}), .., (138.07, \text{devastated})\} . \tag{3.36}$$

Since there is only one feature per example, $P = 1$. The value of K, which is the dimension of the target variable in examples, depends on the encoding we choose. Using a label encoding, the data set would read;

$$\mathcal{D} = \{(22.01, 1), (44.28, 1), .., (138.07, 3)\} , \tag{3.37}$$

so that $K = 1$. This encoding is useful for visualising the dataset, as shown in Fig. 3.4.

For a one-hot encoding,

$$\mathcal{D} = \{(22.01, (1, 0, 0)^T), (44.28, (1, 0, 0)^T), .., (138.07, (0, 0, 1)^T)\} . \tag{3.38}$$

In this case, which we will use for building models, $K = 3$.

The design matrix for this data set is thus a (61×2) matrix:

$$\mathbf{X} = \begin{pmatrix}1 & 22.01\\1 & 44.28\\... & ...\\1 & 138.07\end{pmatrix} . \tag{3.39}$$

For the label encoding, the target matrix, Y, is (61×1), i.e., it is simply a vector;

$$\mathbf{Y} = \begin{pmatrix}1\\1\\...\\3\end{pmatrix} . \tag{3.40}$$

For the vector encoding, Y is (61×3);

$$\mathbf{Y} = \begin{pmatrix}1 & 0 & 0\\1 & 0 & 0\\... & ... & ...\\0 & 0 & 1\end{pmatrix} . \tag{3.41}$$

From this target matrix example, we can see that we don't need to worry too much about the vector-space structure of the one-hot encoding; ultimately, for K categories, we just have a matrix of K entries for each example, where we fill in a "1" at the position (column) corresponding to the label encoding of the category, and zeros everywhere else.

Probabilistic Models for Binary Classification

So far, we have considered building a probabilistic model for regression, i.e. for $p(\mathbf{y}|\mathbf{x})$ where $\mathbf{y}$ is a continuous variable. In categorisation, $\mathbf{y}$ is now a discrete variable. For label encoding, it is a single variable that takes K values, e.g. $k = 1, 2, ..., K$ or $k = 0, 1, 2, ..., K - 1$, which is often more convenient. For one-hot encoding it is a set of K binary variables (one of which is a 1 to indicate the class, the rest being 0).

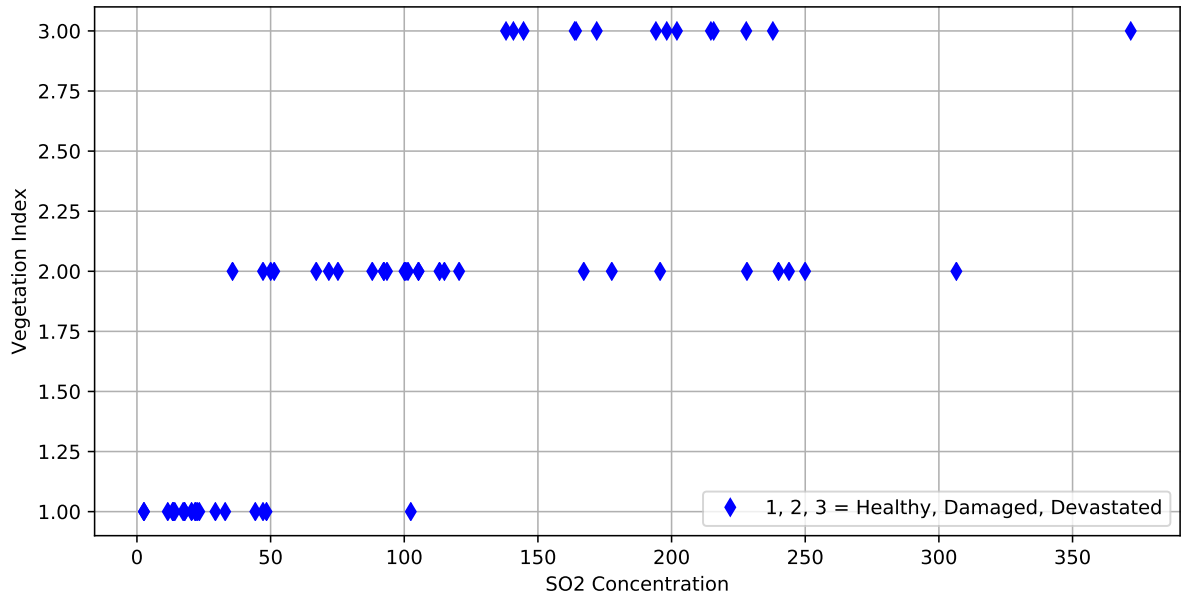

Figure 3.4: Vegetation Index with SO_2 Concentration

For the moment, we will consider a binary classification task, meaning that there are just two categories. In this case, we can use a simple label encoding without too many issues. Therefore, we have a single binary variable, i.e. $K = 1$ and $y = 0, 1$ only.

In the case of binary classification, things are relatively simple. To build a probabilistic model we need to only specify a single function of $\mathbf{x}$, $p(y = 1|\mathbf{x})$, because normalisation tells us that

$$p(y = 0|\mathbf{x}) = 1 - p(y = 1|\mathbf{x}) \ , \tag{3.42}$$

which then completes the specification of the model for all values of y and $\mathbf{x}$.

To write down a model in a single equation, we can make use of indicator functions;

$$p(y|\mathbf{x}) = \mathcal{I}(y = 0)p(y = 0|\mathbf{x}) + \mathcal{I}(y = 1)p(y = 1|\mathbf{x}) \ , \tag{3.43}$$

where $\mathcal{I}(s)$ is one of the statement s is true and zero otherwise. Since we have conveniently chosen the values of y to be zero and one, this expression is in fact the same as

$$p(y|\mathbf{x}) =(1 - y)p(y = 0|\mathbf{x}) + yp(y = 1|\mathbf{x}) \ . \tag{3.44}$$

Alternatively, we can express $p(y|\mathbf{x})$ as a product;

$$p(y|\mathbf{x}) = p(y = 0|\mathbf{x})^{\mathcal{I}(y=0)}p(y = 1|\mathbf{x})^{\mathcal{I}(y=1)} \ , \tag{3.45}$$

$$= p(y = 0|\mathbf{x})^{1-y}p(y = 1|\mathbf{x})^{y} \ , \tag{3.46}$$

which can be confirmed by explicitly substituting in $y = 0$ and $y = 1$.

The advantage of writing $p(y|\mathbf{x})$ as a product like above is that it reveals the structure of the binary classification models we are considering. Recall that we could view building a probabilistic model for regression as considering a function of the example input, $\mu(\mathbf{x})$, which was then chosen to be the mean of the probability distribution (density) we select, which in the previous case was a Gaussian. The Gaussian Linear model was then completed by choosing the function $\mu(\mathbf{x}) = \boldsymbol{\beta} \cdot \mathbf{x}$.

For categorisation of a binary variable, $y = \{0, 1\}$, we again consider a function of the example's input, $\mu(\mathbf{x})$. This is then chosen to be the mean of a distribution. For categorisation, an appropriate choice is the Bernoulli distribution, Ber, i.e.;

$$p(y|\mathbf{x}) = \text{Ber}(y; \mu(\mathbf{x})). \tag{3.47}$$

A Bernoulli distribution describes the simple situation where there are only two possible outcomes; success or failure. In this case, the mean of the distribution is simply equal to the probability of success, i.e., $\mu(\mathbf{x}) = p(y = 1|\mathbf{x})$. On a single trial, the number of successful trials is simply equal to the value of y, while the number of failures is $1 - y$. Therefore, the probability of y can be written as

$$p(y|\mathbf{x}) = p(y = 0|\mathbf{x})^{1-y} p(y = 1|\mathbf{x})^{y} \ . \tag{3.48}$$

Logistic Regression for Binary Categorisation

A natural way to proceed with a building a model for binary classification is as follows; first we choose the Bernoulli distribution. Then, just as in regression, we first specify $\mu(\mathbf{x})$ by combining our features linearly, i.e., as $\boldsymbol{\beta}^T\mathbf{x}$. However, in this case to construct $\mu(\mathbf{x})$, because for binary classification it is equal to $p(y = 1|\mathbf{x})$, we need an extra step to ensure the function is between 0 and 1.

A common choice is to use a sigmoid function, S, so that,

$$p(y = 1|\mathbf{x}; \boldsymbol{\beta}) = S\left(\boldsymbol{\beta}^T\mathbf{x}\right) \ , \tag{3.49}$$

$$= \frac{1}{1 + \exp\left(-\boldsymbol{\beta}^T\mathbf{x}\right)} \ . \tag{3.50}$$

Often, the sigmoid function is denoted as σ, though we will use an S to avoid confusion with the standard deviation.

Using a sigmoid function applied to a linear model of the features in this way is often called logistic regression.

Note that in this model, the only parameters are the linear weights $\boldsymbol{\beta}$, i.e., $\theta = \{\boldsymbol{\beta}\}$.

3.4 Maximum Likelihood Estimation for Classification

Negative Log-Likelihood for Binary Classification

With our model defined, we can now calculate our loss-function. This is given by first calculating the NLL, which will be our loss, and then substituting in the form of our model's predictions to form the loss-function, $L(\theta)$. We then minimise this to find θ^*.

Before considering the logistic model above or indeed any specific model, we can first write down an expression for the NLL for any binary-variable classification problem, i.e. where $y = 0, 1$.

Using the expression for $p(y|\mathbf{x})$ in terms of a product of probabilities, it is possible to split up the log-probabilities appearing in the sum;

$$\text{NLL} = -\frac{1}{N}\sum_{i=1}^{N} \log\left[p\left(y = y^{(i)}|\mathbf{x}^{(i)}\right)\right] \tag{3.51}$$

$$= -\frac{1}{N}\sum_{i=1}^{N} \log\left[p(y = 0|\mathbf{x}^{(i)})^{1-y^{(i)}} p(y = 1|\mathbf{x}^{(i)})^{y^{(i)}}\right] \ , \tag{3.52}$$

$$= -\frac{1}{N}\sum_{i=1}^{N} (1 - y^{(i)}) \log\left[p(y = 0|\mathbf{x}^{(i)})\right] + y^{(i)} \log\left[p(y = 1|\mathbf{x}^{(i)})\right] \ , \tag{3.53}$$

$$= -\frac{1}{N}\sum_{i=1}^{N} (1 - y^{(i)}) \log\left[1 - p(y = 1|\mathbf{x}^{(i)})\right] + y^{(i)} \log\left[p(y = 1|\mathbf{x}^{(i)})\right] \ , \tag{3.54}$$

$$= L \ . \tag{3.55}$$

Loss-Function for Logistic Regression

In the case of a logistic regression model, the form of the NLL is thus;

$$\text{NLL} = -\frac{1}{N}\sum_{i=1}^{N}(1-y^{(i)})\log\left[1-\frac{1}{1+\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}\right]+y^{(i)}\log\left[\frac{1}{1+\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}\right] , \tag{3.56}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(1-y^{(i)})\log\left[\frac{\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}{1+\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}\right]-y^{(i)}\log\left[1+\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\right] \tag{3.57}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(1-y^{(i)})(-\boldsymbol{\beta}^T\mathbf{x}^{(i)})-\log\left[1+\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\right] \tag{3.58}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(1-y^{(i)})(-\boldsymbol{\beta}^T\mathbf{x}^{(i)})-\log\left[\exp\left(-\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\left(\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)+1\right)\right] \tag{3.59}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}(1-y^{(i)})(-\boldsymbol{\beta}^T\mathbf{x}^{(i)})+\boldsymbol{\beta}^T\mathbf{x}^{(i)}-\log\left[\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)+1\right] \tag{3.60}$$

$$= -\frac{1}{N}\sum_{i=1}^{N}y^{(i)}\boldsymbol{\beta}^T\mathbf{x}^{(i)}-\log\left[1+\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\right] , \tag{3.61}$$

$$= L(\boldsymbol{\beta}) . \tag{3.62}$$

For a fixed $\mathcal{D}$, the NLL is a function only of the parameter vector $\boldsymbol{\beta}$. This is our loss function, $L(\boldsymbol{\beta})$ to be minimised to find $\boldsymbol{\beta}^*$.

3.5 Logistic Regression Model for Binary Vegetation Damage

To adapt the vegetation damage example to the case of binary classification, we will consider only two of the three vegetation classes; healthy and devastated. Encoding these as labels, 0 and 1 respectively, we can understand the label as an answer of "true" of "false" to the question "is the vegetation devastated?", with healthy being the only form of non-devastated vegetation by assumption.

Removing the examples of damaged vegetation from the full data set, we are left with $N = 34$ examples, the first few of which are:

$$\mathcal{D} = \{(22.01, 0), (44.28, 0), ..., (138.07, 1)\} . \tag{3.63}$$

The design matrix is (34×2);

$$\mathbf{X} = \begin{pmatrix} 0 & 22.01 \\ 0 & 44.28 \\ ... & ... \\ 1 & 138.07 \end{pmatrix} , \tag{3.64}$$

while the target matrix is size (34×1);

$$\mathbf{Y} = \begin{pmatrix} 0 \\ 0 \\ ... \\ 1 \end{pmatrix} . \tag{3.65}$$

As with the linear models, we could perform a basis expansion to extend the design matrix further. However, we will stick with the basic design matrix for now such that the parameter vector is,

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} . \tag{3.66}$$

The vector for the model's predicted probability of finding devastated vegetation for each example in the

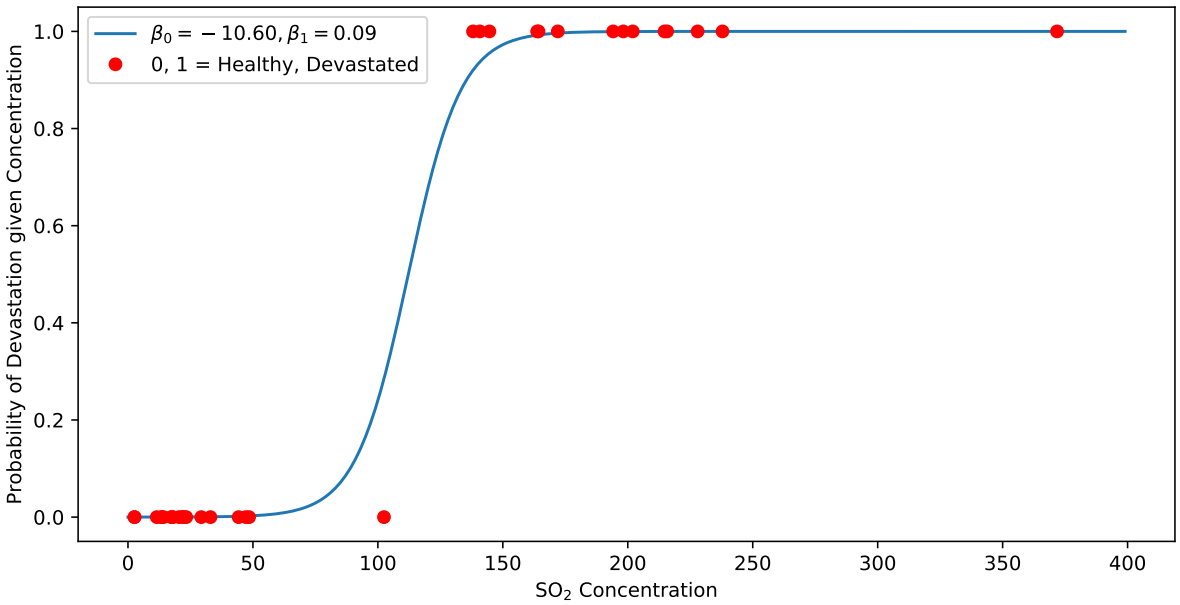

Figure 3.5: Logistic model for the devastation probability given concentration

data set, which we will denote as $\hat{\mathbf{p}}$, is calculated by first taking a linear combination of features;

$$\mathbf{X}\boldsymbol{\beta} = \begin{pmatrix} 1 & 22.01 \\ 1 & 44.28 \\ \dots & \dots \\ 1 & 138.07 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \tag{3.67}$$

$$= \begin{pmatrix} \beta_0 + 22.01\beta_1 \\ \beta_0 + 44.28\beta_1 \\ \dots \\ \beta_0 + 138.07\beta_1 \end{pmatrix} . \tag{3.68}$$

The sigmoid function is then applied to each of these individually to calculate the predicted probabilities that $p(y = 1|\mathbf{x})$, for each example. In other words, we apply the sigmoid function *element-wise* to the vector $\mathbf{X}\boldsymbol{\beta}$. Therefore:

$$\hat{\mathbf{p}} = S\left(\begin{pmatrix} \beta_0 + 22.01\beta_1 \\ \beta_0 + 44.28\beta_1 \\ \dots \\ \beta_0 + 138.07\beta_1 \end{pmatrix}\right) , \tag{3.69}$$

$$= \begin{pmatrix} S(\beta_0 + 22.01\beta_1) \\ S(\beta_0 + 44.28\beta_1) \\ \dots \\ S(\beta_0 + 138.07\beta_1) \end{pmatrix} . \tag{3.70}$$

As an example, we can choose the values $\boldsymbol{\beta} = (-10.6, 0.09)$. The predicted probabilities of $y = 1$ or various values of SO_2 concentrations, i.e. values of x_1, are shown in Fig 3.5, along with the data for comparison.

Finally, we can calculate the negative log-likelihood. This is illustrated by slices, i.e. as a function of β_0 and β_1 for fixed β_1 and β_0 respectively, see Fig. 3.6

3.6 Minimisation of the NLL with Gradient Descent

Gradient Descent for NLL Minimisation

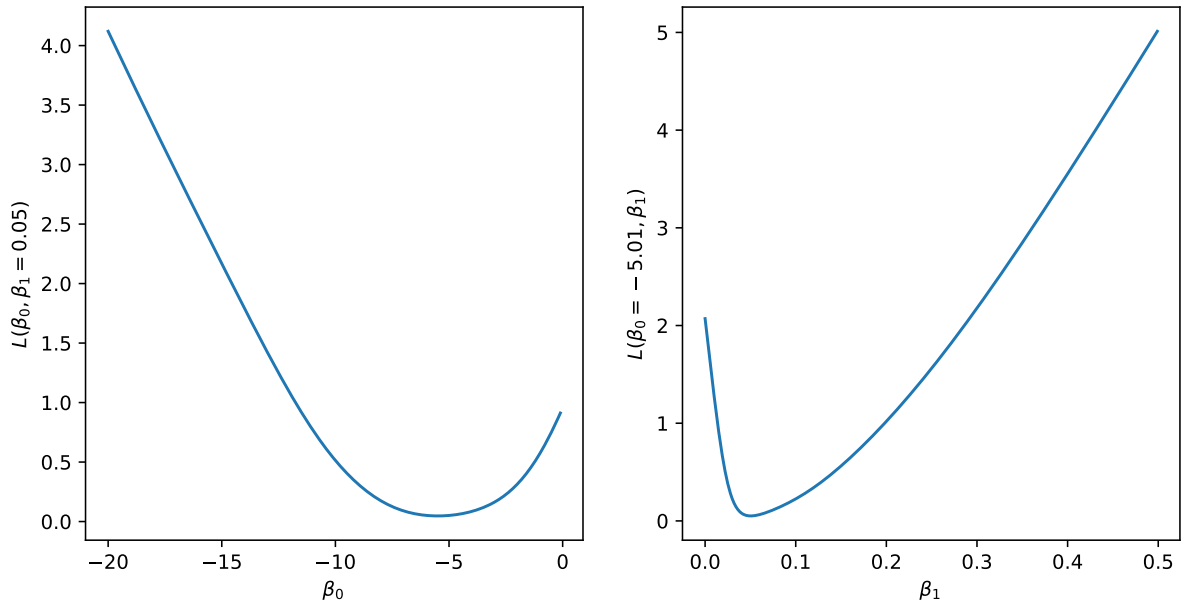

Figure 3.6: Behaviour of the Negative Log-Likelihood for a Logistic Model of Vegetation Index. From the plots, we could estimate that the optimal choices for the parameters – where the loss function is at its lowest – appear to be around $\beta_0 \approx -5$ and $\beta_1 \approx 0.05$. However, to produce the plots we have fixed one of the values of the parameters in each case, so this might be misleading. To minimise the loss function correctly we must consider varying both values simultaneously to find the minimum.

The minimisation problem we want to perform for the logistic regression model is of a different character to that of the Gaussian linear model because the parameters appear inside a non-linear function. In other words, the problem is non-linear, and we will not be able to solve it analytically as before. In fact, this is the general case, and we can only solve optimisation problems exactly for a limited number of models. Therefore, we require a numerical strategy for solving such problems.

To minimise the NLL numerically, we want to devise a systematic way to, starting from some initial guess for the parameters, update this guess so that the loss function becomes smaller. In this case, we know that the new guess produced is superior, and iterating this procedure will lead to, in principle, better and better estimates for the optimal choice of parameters.

Considering the case where our parameters are encoded in a vector, $\boldsymbol{\beta}$, we proceed by choosing some initial guess for the values. We then update this guess by adding to it some other vector, $\boldsymbol{v}$, which produces a new guess, $\boldsymbol{\beta}'$, as;

$$\boldsymbol{\beta} \to \boldsymbol{\beta}' = \boldsymbol{\beta} + \boldsymbol{v} \ . \tag{3.71}$$

Of course, we must choose $\boldsymbol{v}$ appropriately so that the new guess is better than the previous one, in the sense that the value of the loss-function is lower, or at worst equal;

$$L(\boldsymbol{\beta}') \leq L(\boldsymbol{\beta}) \ . \tag{3.72}$$

To develop an algorithm to achieve this, we start by rewriting the vector $\boldsymbol{v}$ as equal to a unit vector, $\boldsymbol{u}$, for which $\boldsymbol{u} \cdot \boldsymbol{u} = 1$, multiplied by some number, α, i.e., $\boldsymbol{v} = \alpha \boldsymbol{u}$, and;

$$\boldsymbol{\beta} \to \boldsymbol{\beta}' = \boldsymbol{\beta} + \alpha \boldsymbol{u} \ . \tag{3.73}$$

Expressing our updates in this manner is convenient, because we will be able to choose α separately to determine how large a change we should make to $\boldsymbol{\beta}$.

All that remains is then to determine $\boldsymbol{u}$, i.e., which direction we should update our parameter vector in to decrease the value of the loss function.

A sensible method for doing this is to pick the unit vector which is the best-choice, in the sense that it will lead to the fastest decrease in $L(\boldsymbol{\beta})$. To quantify this idea, we use the concept of a *directional derivative*.

For a function of a vector, such as $L(\boldsymbol{\beta})$, the directional derivative is defined as the derivative of the function when adding to the argument vector another vector in a particular direction defined by a unit vector $\boldsymbol{u}$. Therefore, for any potential direction – corresponding to a choice of unit vector $\boldsymbol{u}$ – we can ask how quickly the function changes in that direction via;

$$D_{\boldsymbol{u}}(L(\boldsymbol{\beta})) = \frac{\partial L(\boldsymbol{\beta} + \alpha \boldsymbol{u})}{\partial \alpha}|_{\alpha=0} \tag{3.74}$$

To express the directional derivative in a more convenient and informative manner, we can vectorise the expression as follows, using the chain-rule for multivariate functions;

$$\begin{aligned} D_{\boldsymbol{u}}(L(\boldsymbol{\beta})) &= \frac{\partial L(\beta_0 + \alpha u_0, \beta_1 + \alpha u_1, \beta_2 + \alpha u_2, ...,)}{\partial \alpha}|_{\alpha=0} && (3.75)\\ &= \frac{\partial L}{\partial(\beta_0 + \alpha u_0)}\frac{\partial(\beta_0 + \alpha u_0)}{\partial \alpha} + \frac{\partial L}{\partial(\beta_1 + \alpha u_1)}\frac{\partial(\beta_1 + \alpha u_1)}{\partial \alpha} + ...|_{\alpha=0} && (3.76)\\ &= \frac{\partial L}{\partial(\beta_0 + \alpha u_0)}u_0 + \frac{\partial L}{\partial(\beta_1 + \alpha u_1)}u_1 + ...|_{\alpha=0} && (3.77)\\ &= \sum_{p=0}^{P} \frac{\partial L}{\partial(\beta_p + \alpha u_p)}u_p|_{\alpha=0} \ . && (3.78) \end{aligned}$$

The first factor is just the components of the vector gradient of the loss with respect to $\boldsymbol{\beta} + \alpha \boldsymbol{u}$,

$$\begin{aligned} D_{\boldsymbol{u}}(L(\boldsymbol{\beta})) &= \sum_{p=0}^{P} \left[\nabla_{\boldsymbol{\beta}+\alpha\boldsymbol{u}} L\right]_p u_p|_{\alpha=0} \ , && (3.79)\\ &= (\nabla_{\boldsymbol{\beta}+\alpha\boldsymbol{u}} L) \cdot \boldsymbol{u}|_{\alpha=0} \ , && (3.80)\\ &= \boldsymbol{u}^T \left(\nabla_{\boldsymbol{\beta}+\alpha\boldsymbol{u}} L\right)|_{\alpha=0} \ , && (3.81)\\ &= \boldsymbol{u}^T \left(\nabla_{\boldsymbol{\beta}} L\right) \ . && (3.82) \end{aligned}$$

So, the directional derivative in a direction $\boldsymbol{u}$ is simply the vector gradient of the function, dotted with the unit vector in question. In this sense, it just tells us the overall rate of change of the function along the chosen direction.

With the vectorised expression for the directional derivative, we can now consider picking the optimal choice of $\boldsymbol{u}$, so that our updates to $\boldsymbol{\beta}$ decrease the loss function as quickly as possible. This means solving the optimisation problem;

$$\begin{aligned} \boldsymbol{u}^* &= \arg\min_{\boldsymbol{u}} D_{\boldsymbol{u}}(L) \ , && (3.83)\\ &= \arg\min_{\boldsymbol{u}} \boldsymbol{u}^T \left(\nabla_{\boldsymbol{\beta}} L\right) \ . && (3.84) \end{aligned}$$

To solve this problem, we express the dot product of the two vectors as the product of the lengths of the two vectors and the cosine of the angle between them;

$$\begin{aligned} \boldsymbol{u}^T \left(\nabla_{\boldsymbol{\beta}} L\right) &= ||\boldsymbol{u}||_2 || \left(\nabla_{\boldsymbol{\beta}} L\right)||_2 \cos\left(\phi\right) \ , && (3.85)\\ &= || \left(\nabla_{\boldsymbol{\beta}} L\right)||_2 \cos\left(\phi\right) \ , && (3.86) \end{aligned}$$

where the second line follows because $\boldsymbol{u}$ is a unit vector. With this expression, we can now solve the minimisation problem, because the length of a vector is always positive, and all that remains is to choose the angle ϕ. The minimum of a cosine is at $\phi = \pi$. In other words, the unit vector must point in the opposite direction to the gradient vector. Therefore, the optimal unit vector is;

$$\boldsymbol{u}^* = -\frac{\left(\nabla_{\boldsymbol{\beta}} L\right)}{|| \left(\nabla_{\boldsymbol{\beta}} L\right)||_2} \ . \tag{3.87}$$

With this, we come to the fairly intuitive result that we should update our parameter vector according to;

$$\boldsymbol{\beta} \to \boldsymbol{\beta}' = \boldsymbol{\beta} - \alpha \left(\nabla_{\boldsymbol{\beta}} L\right) \ . \tag{3.88}$$

Such an update scheme is known as gradient descent, and is the standard way to minimise loss functions, one that with some adjustments can be applied to even highly sophisticated machine learning algorithms.

Gradient Descent for Logistic Regression

To apply gradient descent to Logistic regression, we calculate the (components of) the gradient with respect to the parameters as usual.

The components of the gradient are;

$$\begin{aligned}
\left(\nabla_{\boldsymbol{\beta}} L\right)_p &= \frac{\partial L}{\partial \beta_p} && (3.89)\\
&= -\frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial \beta_p}\left[y^{(i)}\boldsymbol{\beta}^T\mathbf{x}^{(i)} - \log\left[1+\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\right]\right] && (3.90)\\
&= \frac{-1}{N}\sum_{i=1}^{N} x_p^{(i)}\left[y^{(i)} - \frac{\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}{1+\exp\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)}\right] && (3.91)\\
&= \frac{-1}{N}\sum_{i=1}^{N} x_p^{(i)}\left[y^{(i)} - S\left(\boldsymbol{\beta}^T\mathbf{x}^{(i)}\right)\right] \ . && (3.92)
\end{aligned}$$

In terms of the design and target matrix notation the gradient can be expressed as;

$$\begin{aligned}
\left(\nabla_{\boldsymbol{\beta}} L\right)_p &= -\frac{1}{N}\sum_{i=1}^{N} X_{ip}\left(Y_i - S\left(\left[\mathbf{X}\boldsymbol{\beta}\right]_i\right)\right) && (3.93)\\
&= -\frac{1}{N}\sum_{i=1}^{N} X_{ip}\left(Y_i - \left[S\left(\mathbf{X}\boldsymbol{\beta}\right)\right]_i\right) && (3.94)\\
&= -\frac{1}{N}\sum_{i=1}^{N} X_{ip}\left[\boldsymbol{Y} - S\left(\mathbf{X}\boldsymbol{\beta}\right)\right]_i && (3.95)\\
&= -\frac{1}{N}\sum_{i=1}^{N} \left[\mathbf{X}^T\right]_{pi}\left[\boldsymbol{Y} - S\left(\mathbf{X}\boldsymbol{\beta}\right)\right]_i && (3.96)\\
&= -\frac{1}{N}\left[\mathbf{X}^T\left(\boldsymbol{Y} - S\left(\mathbf{X}\boldsymbol{\beta}\right)\right)\right]_p \ , && (3.97)
\end{aligned}$$

where in the second line we should again understand the sigmoid function as being applied element-wise to the vector arguments.

With this calculation, we have that;

$$\begin{aligned}
\nabla_{\boldsymbol{\beta}} L &= -\frac{1}{N}\mathbf{X}^T\left(\boldsymbol{Y} - S\left(\mathbf{X}\boldsymbol{\beta}\right)\right) && (3.98)\\
&= \frac{1}{N}\mathbf{X}^T\left(S\left(\mathbf{X}\boldsymbol{\beta}\right) - \boldsymbol{Y}\right) \ . && (3.99)
\end{aligned}$$

MLE for Logistic Regression Model of Vegetation Damage using Gradient Descent

For any value of $\boldsymbol{\beta}$, we can evaluate the gradient of the loss function as;

$$\nabla_{\boldsymbol{\beta}} L = \frac{1}{N}\left[\begin{pmatrix} 1 & 1 & ... & 1 \\ 22.01 & 44.28 & ... & 128.07 \end{pmatrix}\begin{pmatrix} S(\beta_0 + 22.01\beta_1) - 0 \\ S(\beta_0 + 44.28\beta_1) - 0 \\ ... \\ S(\beta_0 + 138.07\beta_1) - 1 \end{pmatrix}\right] \ . \tag{3.100}$$

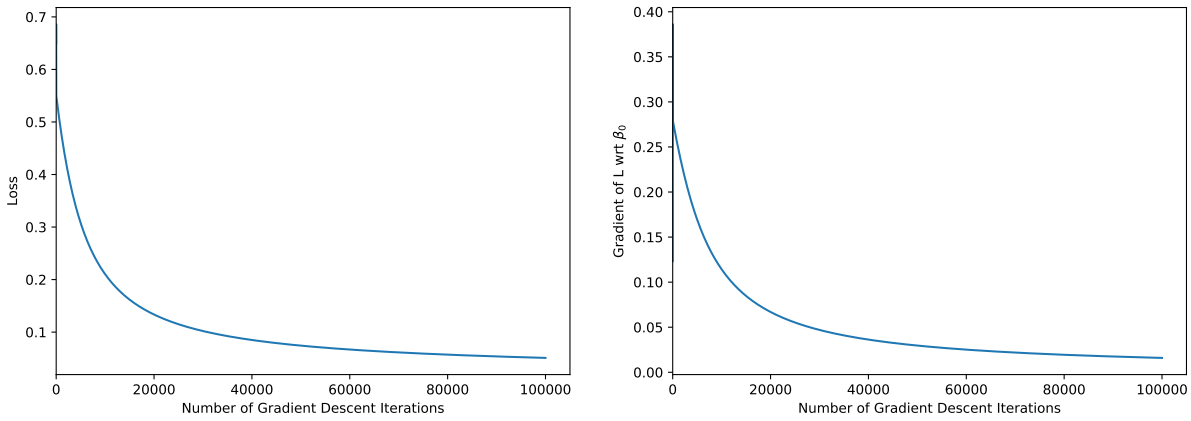

Figure 3.7: Behaviour of NLL under Gradient Descent for the Vegetation Damage Classification and logistic regression. The left-hand plot shows how the loss decreases with each successive update to $\boldsymbol{\beta}$. The right-hand plot shows how the gradient of the loss changes under gradient descent. As the update is improved, the gradient of the loss becomes smaller, vanishing at the minimum itself. In this way, the gradient descent naturally decreases the size of updates on approach to the desired minimum.

To perform gradient descent, we start with a randomly chosen pair of values, and update $\boldsymbol{\beta}$ according to;

$$\boldsymbol{\beta} \to \boldsymbol{\beta} - \alpha \nabla_{\boldsymbol{\beta}} L \ , \tag{3.101}$$

i.e.

$$\begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} - \alpha \frac{1}{N} \left[\begin{pmatrix} 1 & 1 & ... & 1 \\ 22.01 & 44.28 & ... & 128.07 \end{pmatrix} \begin{pmatrix} S(\beta_0 + 22.01\beta_1) - 0 \\ S(\beta_0 + 44.28\beta_1) - 0 \\ ... \\ S(\beta_0 + 138.07\beta_1) - 1 \end{pmatrix} \right] \ . \tag{3.102}$$

The behaviour of the NLL under gradient descent in shown in Fig. 3.7.

Chapter 4

Generalisation, Overfitting and Regularisation

4.1 Generalisation Error

Typically, in supervised learning problems, our task is to develop a model that can provide predictions on previously unseen data (e.g. classify new images). This is quite different from simply describing examples that have already been presented (i.e. in SL for which the "answer", i.e. the label of the target, is already known to our model). Indeed, we do not just want to learn about the data we have but to learn from it, in order to generalise from this data to new, previously unseen cases. This is the concept of *generalisation*, which is in reality the task that we almost always want to perform in SL.

Since the task we are primarily interested in is generalisation, it is important to choose a performance measure that reflects this. However, here we encounter an issue: so far the performance measure has been our guide for developing our models, e.g. for choosing the best-values of our model's parameters, which can only be done using data (experience). Therefore, in SL problems there is a tension between the ability to describe a data set – with performance measured on the data set which is used to establish the model – and generalisation – with performance measured on previously unseen examples, i.e. those not used to establish the model.

To understand this issue, consider a simple regression scenario where our performance measure is the squared-loss as before;

$$L = \frac{1}{M}\sum_{j=1}^{M}(\hat{y}^{(j)} - y^{(j)}) \ ,$$

where now we haven't specified exactly which examples to consider yet, just the form of our per-example loss (the squared error in this case) and that L is given by averaging the per-example loss over some set of examples.

In order to establish our model, we would proceed as before and consider examples taken from our data set. In this case, the loss is often called the "training error", since the data set is used to "train" the model to improve performance;

$$L_{\text{train}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)}) \ . \tag{4.1}$$

When we want to be explicit that the data set, $\mathcal{D}$, is used for training the model we can use the notation $\mathcal{D}_{\text{train}}$ instead of simply $\mathcal{D}$, along with N_{train} instead of N. However, typically we will simply refer to the data set, $\mathcal{D}$, which should be understood as the set of examples used for training.

While $\mathcal{D}_{\text{train}}$ is used to develop our model, to really measure the performance of our model on the task that we are interested in (generalisation), we should consider a set of unseen examples. To quantify the

performance on generalisation, we can then evaluate the chosen loss on the unseen examples. This is often called the *generalisation-error* or *test-error*;

$$L_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} (\hat{y}^{(j)} - y^{(j)}) \ . \tag{4.2}$$

Here, N_{test} is the number of examples in the "test-set", denoted $\mathcal{D}_{\text{test}}$, which is the data set containing examples that are used to evaluate performance, but not to directly establish the model via fitting.

In practice, a simple way to proceed is to take the data set once has, $\mathcal{D}$, and split it (randomly) into two separate data sets, $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$. The amount of examples in each set will depend on the problem in hand, though $\mathcal{D}_{\text{train}}$ should be considerably larger than $\mathcal{D}_{\text{test}}$, i.e. $N_{\text{train}} \gg N_{\text{test}}$. For example, one might use an $80:20$ split with 80% of the data in $\mathcal{D}$ randomly chosen to make up $\mathcal{D}_{\text{train}}$ and the remaining examples going to $\mathcal{D}_{\text{test}}$. The data contained in $\mathcal{D}_{\text{train}}$ then goes into L_{train}, which is minimised to find the values of the model's parameters, θ^*. The examples from $\mathcal{D}_{\text{test}}$ are instead used in L_{test}, which is not used to determine the model's parameters, but as a measure of performance only.

Note that we can also repeat this idea of other performance measures that are not the loss (i.e. the "metrics" we are interested in but are not used explicitly for fitting). Like losses, metrics (e.g. the accuracy of our classification model) can be evaluated on both the training and test data to measure performance, with the latter measuring how well the model is generalising for this particular metric.

4.2 Overfitting and Model Complexity

In practice, two issues can arise with machine learning algorithms that reflects the tension between establishing a model using a data set and generalisation error. The first is underfitting: In this scenario, a model is not sufficiently complex to describe the provided data set. For example, one might pick a very restricted parametric model to describe a complicated data set e.g. a simple straight line for clearly non-linear data. In practice, one can often make models as complicated as required using flexible modelling frameworks such as neural networks (typically now referred to as "deep learning").

The second potential issue is that of overfitting: In this case the model is too specialised to the data set and fails to generalise. This is the case when model complexity is too high for a data set. For instance, if data is roughly linear then a high order polynomial model will certainly fit the data, but will likely contain many unnecessary "wiggles" that mean it generalises poorly.

Both underfitting and overfitting are typically due to a poor balance of model complexity versus the amount of data available. This is familiar from the simple case of fitting a polynomial curve to data. If the order of the polynomial is very high, then we will be able to describe simple data sets perfectly, though the curve will tend to behave wildly and thus cannot be expected to describe data points to which it has not been fit. This is overfitting. Including more data points will help alleviate overfitting, assuming the final curve is sufficiently complicated. On the other hand, a very low-order polynomial (e.g. a straight line) is unlikely to provide a good fit to complex data sets that contain many data points, an example of under-fitting.

To quantify the ideas of underfitting and overfitting, we can use the losses evaluated on our training and test sets, L_{train} and L_{test}. When the training-loss is much larger than the performance we require for a particular task, then our model is underfitting. For example, on a task of image recognition for medical applications, the required performance may be the success rate of a human doctor in classifying the chosen images. If the model has a performance that is much worse than this on $\mathcal{D}_{\text{train}}$, then we can say it underfits.

Overfitting is then quantified by the difference between L_{test} and L_{train}. Since L_{test} is always evaluated on unseen examples, it will be larger than L_{train}. However, if it is much larger than we know that the model's predictions on unseen examples are considerably worse than on the seen examples, i.e., the model overfits.

4.3 Polynomial Basis Expansions for Arctic Ice Regression

If we quantify the complexity of a model by some value λ, which we are able to change, then a typical behaviour of L_{train} and L_{test} as we increase λ from an initially small value (simple model) is as follows:

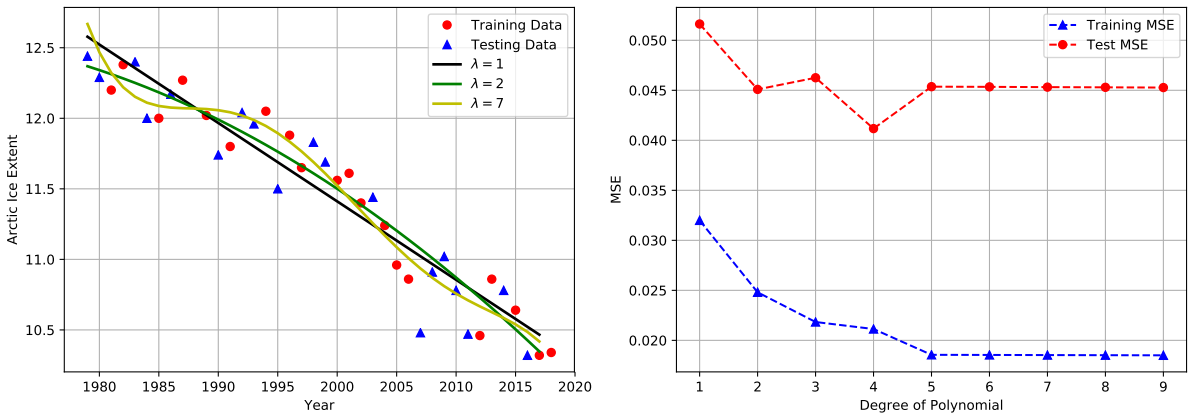

Figure 4.1: The left-hand plot shows various models fit to the Arctic ice extent data (mean extent per year). In this case, the increasing complexity of the models corresponds to the degree of the polynomial, λ, which manifests itself in more local minima/maxima in the corresponding curves. The right-hand plot displays the training and testing error as a function of the model complexity.

First, both L_{train} and L_{test} should increase with λ. This shows that the model is becoming sufficiently complex for the problem in hand. As λ increases further, L_{train} will continue to decrease as a more complex model will always improve the fit to a given data set. However, after some point, L_{test} will begin to increase as the model becomes overly specialised to the data, i.e. it overfits.

This idea is shown in Fig. 4.1 for a few example Linear Models with a polynomial basis expansion for the Arctic ice problem. In this case, for illustration, half of the points have been selected at random to be the training set, and half to be the test set. The loss was chosen to be the MSE as before, and is evaluated separately on the training and test sets. The degree of the chosen polynomial is used as a measure of the complexity in this case, since higher degree polynomials are more general than lower degree ones, and can ultimately be used to describe any function via a Taylor expansion.

4.4 Weight Decay Regularisation

The concept of overfitting leads us to, in addition to optimisation, a second key pillar of modern machine learning; regularisation. This can be defined as follows: "Regularisation is any modification we make to a learning algorithm that is intended to reduce its generalisation error but not its training error" (Goodfellow 2016).

In the polynomial basis expansion example, reducing the degree of the polynomial would be an example of regularisation. So long as the polynomial does not become too simple, this will not harm the fit to the data, while improving generalisation.

The idea of regularisation is very flexible, and a number of practical schemes have been developed to effectively reduce overfitting. A number of these are implemented by modifying the cost-function to be optimised (such as the MSE) when establishing the model's parameters. In terms of the function to be minimised, which we will denote $J(\theta)$, this means that we have two parts: The usual data-dependent loss term (e.g. the $L(\theta; \mathcal{D}) = \text{MSE}_{\text{train}}$) and the data-independent "regularisation" term $R(\theta)$:

$$J(\theta) = L(\theta; \mathcal{D}) + R(\theta) \ . \tag{4.3}$$

Often, we simply jump to this point and choose a regularisation term. Ultimately, this is a pragmatic approach – we can always estimate the generalisation error using a test-set, and if this is the key concern then we can simply try out different forms of regularisation by choosing $R(\theta)$ and see what works. However, regularisations that take this form, by adding a term to the loss, can also be justified by more fundamental considerations, as we will see.

A commonly applied form of regularisation is known as weight-decay or l2 regularisation. Intuitively, the idea is that, when dealing with a complex model such as a high-degree polynomial, the coefficients (the weights) can become very large (both negative and positive) in order to cancel "in just the right

way" to fit the relatively simple data. This leads to many unnecessary local maxima and minimum that lie far from the data, spoiling the model's generalisation. To remedy this, we can then constrain how large we allow the coefficients to become, so that such large cancellations cannot take place.

A simple way to achieve this is by considering the dot-product of the parameter vector $\boldsymbol{\beta}$ with itself, i.e., $\boldsymbol{\beta} \cdot \boldsymbol{\beta} - \beta_0^2$, where we have subtracted the constant term β_0 as this just provides an overall shift to our model independently of any features, and does not directly impact the complexity.

In the example of a linear model for regression, adding this dot-product to the usual loss then produces a new regularised loss function:

$$J(\theta) = \text{MSE}(\boldsymbol{\beta}; \mathcal{D}) + \kappa \boldsymbol{\beta}^T \boldsymbol{\beta} - \kappa \beta_0^2 \ , \tag{4.4}$$

$$= \text{MSE}(\boldsymbol{\beta}; \mathcal{D}) + \kappa \sum_{p=1}^{P} \beta_p^2 \ , \tag{4.5}$$

where we have also introduced a (hyper)parameter κ that controls how much regularisation we should apply.

As $\kappa \to 0$ the regularisation term becomes less and less important, and we recover our original loss function. Therefore, the model will be just as complex as it was without regularisation. As $\kappa \to \infty$ the opposite is true and the original loss function, in this case $\text{MSE}(\beta; \mathcal{D})$, is irrelevant. Instead, only the R term matters. In this example, that would mean the best model minimises $\sum_{p=1}^{P} \beta_p^2$, i.e., it has all the parameters equal to zero except β_0. This is just the constant model that we have seen before which is essentially the simplest model possible. As such, we can see that the regularisation hyperparameter, κ, controls model complexity, with more complicated models corresponding to smaller values of κ. Therefore, we could consider writing $\lambda = \kappa^{-1}$, since the complexity of our model, λ, is inverse to the size of κ.

Note that while κ or λ are indeed parameters, they are treated quite differently to the other model's parameters θ. In particular, they do not directly correspond to parameters in the functions we used to make predictions, and their "best-values" are not directly determined by fitting them to the training set, which is how we establish θ^*. Therefore, they are often distinguished from other parameters and are called hyperparameters.

Of course, we would also like to select the best-values of our hyperparameters. However, we cannot do this in the same way as the other parameters, by fitting to the training set, otherwise we would run into all the problems of overfitting that we are trying to avoid. Instead, to select values of the hyperparameters other methods must be used. While we will not explore these here, a basic example would be to evaluate the generalisation-error for several values of the hyperparameters, and pick the values that have the lowest error. Even better, one could further subdivide the test-set into two parts, one "development set" for choosing the hyperparameter values and one true "test-set" for evaluating model performance on which no parameter or hyperparameter values are chosen, thus providing a fair measure of ability to generalise.

Fig. 4.1 shows an example of a linear model with sixth-order polynomial basis expansion for the Arctic ice extent regression problem, where weight-decay regularisation is used.

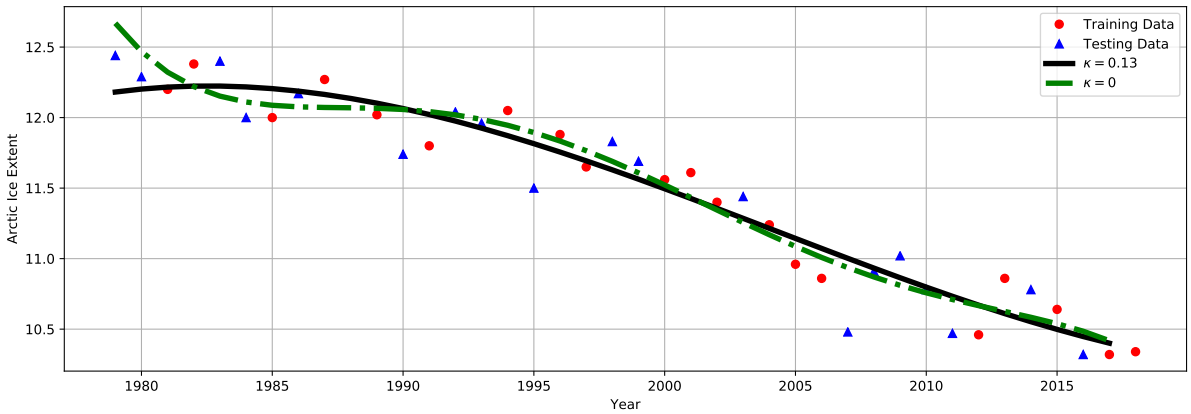

Figure 4.2: Sixth Degree Polynomial Fits of the Arctic Ice, with weight decay. As the value of κ is increased, the model is regularised, which results in a smoothing out of the line of predictions, helping to improve generalisation.

Chapter 5

Bayesian Views of Probabilistic Models

Before turning to the problem of developing a flexible framework for high complexity models – known as neural networks – we can gain a deeper understanding of regularisation by considering a Bayesian view of probabilities. In fact, with the Bayesian viewpoint, we will be able to gain not just a deeper understanding of regularisation, but an appreciation for the link between the probabilistic and deterministic models we have developed (via Bayesian decision theory) and a better justification for the use of maximum likelihood estimation for training.

5.1 Bayesian Parameter Estimation: The Posterior

When constructing probabilistic models for supervised learning, our interest is in modelling $p(y|x)$. So far, we have been constructing parametrised models, $p(y|x;\theta)$, where here θ simply parametrises a set of functions that we might use for predictions. To fix the value of θ we fit to our (training) data set, which fixes a value θ^*. In this way, we end up with a model with no remaining parameters, $p(y|x)$, which can be used for prediction as we wished.

However, rather than picking just a single value for our parameters, we can also consider quantifying our uncertainty about the parameter θ using probabilities. In this case, the parameters of the model can be treated in just the same way as other variables, such as the input x. In terms of notation, rather than write $p(y|x;\theta)$ which indicated that θ is to be treated differently from other quantities, we can write $p(y|x,\theta)$. In this second notation, both the input x and the parameters θ appear as quantities that are to be conditioned on, i.e., that we used as our assumptions when predicting the probabilities for values of y.

In this Bayesian framework, the key object that we want to consider is known as the posterior. In the context of SL, the posterior is the probability of having a particular value of θ, when the data set is used as an input. In other words, we want to reason about the possible values of the parameters based on the data we have available. The posterior is therefore $p(\theta|\mathcal{D})$. In order to compute $p(\theta|\mathcal{D})$, we make use of Bayes' theorem, which relates the posterior to other quantities that can be computed more directly. To come to Bayes' theorem, we use a basic rule of probabilities that converts joint probabilities to conditional probabilities;

$$p(\theta,\mathcal{D}) = p(\theta|\mathcal{D})p(\mathcal{D}) \ , \tag{5.1}$$
$$= p(\mathcal{D}|\theta)p(\theta) \ . \tag{5.2}$$

Equating these two lines and rearranging we get;

$$p(\theta|\mathcal{D}) = p(\mathcal{D})^{-1}p(\mathcal{D}|\theta)p(\theta) \ , \tag{5.3}$$
$$= \mathcal{Z} \ p(\mathcal{D}|\theta)p(\theta) \ , \tag{5.4}$$
$$= \mathcal{Z} \ \mathcal{L}(\theta;\mathcal{D})p(\theta) \ . \tag{5.5}$$

The posterior is therefore made of three factors: The first factor, $\mathcal{Z}$, simply ensures normalisation (probabilities sum to one). The second factor is the probability predicted for the data, given a particular choice of parameters. This is just the likelihood that we have considered previously in the context of MLE. Finally, there is a new factor, $p(\theta)$, known as the prior, which encodes our beliefs about the probability of different values of θ, without consideration of the data. In this sense, Bayes' theorem above can be seen as a way of taking a prior belief about the value of θ, encoded in the prior, and updating it in the face of evidence (the data encoded in the Likelihood) in order to come to a new set of beliefs about the values of θ, encoded in the posterior.

5.2 Bayesian Parameter Estimation: Making Predictions

For a given model, data set and prior probabilities, the posterior can be computed as above. Therefore, we end up with a whole distribution of possible values for θ, in contrast to the single value, θ^*, we had obtained previously. To actually make predictions with this distribution, we can sum over all possible values of θ, i.e., we marginalise as

$$p\left(y|x,\mathcal{D}\right) = \sum_{\theta} p\left(y,\theta|x,\mathcal{D}\right) \ , \tag{5.6}$$

$$= \sum_{\theta} p\left(y|x,\theta,\mathcal{D}\right) p(\theta|x,\mathcal{D}) \ , \tag{5.7}$$

where we have assumed that θ is discrete (for continuous parameters the above sum becomes an integral) and kept the conditioning on the data set explicit (since we always use our experience to determine predictions).

To simplify this expression, we use the fact that the new example's input, x, that we want to make a prediction about is not used to determine the values of the parameters. In other words, $p(\theta|x,\mathcal{D}) = p(\theta|\mathcal{D})$. Additionally, since we are using a parametric model, we will assume that once the data is used to determine the probabilities of parameters, it is not used further in predictions. Therefore, $p\left(y|x,\theta,\mathcal{D}\right) = p\left(y|x,\theta\right)$. With these simplifications, we come to;

$$p\left(y|x,\mathcal{D}\right) = \sum_{\theta} p\left(y|x,\theta\right) p(\theta|\mathcal{D}) \ . \tag{5.8}$$

While the previous expression allows us to make predictions for new examples using the full posterior, it is most often helpful to approximate the posterior in some way so simplify calculations. To do this, we could, for example, simply assume that $p(\theta|\mathcal{D})$ is highly peaked about some value θ^*. In other words, given the data, we have a very strong belief that θ takes the value θ^*. In that case, the above marginalisation actually becomes equivalent to the way we made predictions previously. We take a single value of θ^*, plug it into our models and make predictions with θ fixed to this value. Now we can see that, while convenient, the way we have been making predictions previously for probabilistic models is typically just an approximation, though it might well be sufficient or even necessary to make the calculations possible in practice.

To formalism this idea, we can approximate $p(\theta|\mathcal{D})$ using an indicator function, $\mathcal{I}(\theta = \theta^*)$ (which recall takes the value one if $\theta = \theta^*$ is true or zero otherwise) or alternatively a delta-function about θ^* when dealing with probability densities. In that case:

$$p\left(y|x,\mathcal{D}\right) = \sum_{\theta} p\left(y|x,\theta\right) p(\theta|\mathcal{D}) \ , \tag{5.9}$$

$$\approx \sum_{\theta} p\left(y|x,\theta\right) \mathcal{I}(\theta = \theta^*) \ , \tag{5.10}$$

$$\approx p\left(y|x,\theta^*\right) \ . \tag{5.11}$$

5.3 Bayesian Decision Theory: Deterministic Predictions with Probabilistic Models

Let us now consider what we should do if we have a probabilistic model but need to return a single prediction, $\hat{y}$, as is often the case with machine learning tasks.

The way to approach this is to define a decision-loss function, $D(y, \hat{y})$, that measures how much an incorrect prediction (decision) should be punished. For example, we could choose the usual squared-error,

$$D(y, \hat{y}) = (\hat{y} - y)^2 \ . \tag{5.12}$$

This might, for example, reflect the real-life cost of making an incorrect decision using our machine learning algorithm when a definite description is needed (e.g. given a model for the probability of a patient having some disease, we must make a definite decision to treat it or not, and should be informed by the possible associated costs to health of an incorrect decision).

For probabilistic models, we have a distribution over possible y values, and we can calculate the expected decision-loss with respect to our model $p(y|x)$:

$$\rho(\hat{y}) = \mathbb{E}_{y \sim p(y|x)} \left[D(\hat{y}, y) \right] \ . \tag{5.13}$$

The "best" single-value prediction we could make is then the value of $\hat{y}$ that minimises this expected loss.

Note that this concept is somewhat different from the minimisation of the loss we have been using so far: The current decision-loss, D, refers to how we should make a single prediction from our model, once the model is established from the data. The other loss we have been using, L, appears as a way to establish the model in the first place, using the data and a method such as MLE.

To make the best-prediction, we thus want to minimise $\rho(\hat{y})$ to give,

$$\hat{y}^* = \text{argmin}_{\hat{y}} \rho(\hat{y}) \ . \tag{5.14}$$

For a discrete variable, this optimal prediction takes the form,

$$\hat{y}^* = \text{argmin}_{\hat{y}} \sum_y D(\hat{y}, y) p(y|x) \ . \tag{5.15}$$

Therefore, for the case of the squared-error decision-loss:

$$\hat{y}^* = \text{argmin}_{\hat{y}} \sum_y (\hat{y} - y)^2 p(y|x) \ , \tag{5.16}$$

$$= \text{argmin}_{\hat{y}} \sum_y (\hat{y}^2 - 2\hat{y}y + y^2) p(y|x) \ , \tag{5.17}$$

$$= \text{argmin}_{\hat{y}} \hat{y}^2 - 2\mathbb{E}_{y \sim p(y|x)}[y]\hat{y} + \mathbb{E}_{y \sim p(y|x)}[y^2] \ . \tag{5.18}$$

By differentiating with respect to $\hat{y}$, we find that the optimal choice of prediction is given by,

$$\hat{y}^* = \mathbb{E}_{y \sim p(y|x)}[y] \ . \tag{5.19}$$

Therefore, for the squared-error decision-loss, the optimal choice we should make is that of the expected y value.

5.4 Maximum a Posteriori (MAP) Estimation

We can view all the principles that we have used previously, which select a single value, as justified by a highly-peaked posterior distribution $p(\theta|\mathcal{D})$ over our parameters. However, even if we do not know the form of the posterior in order to ascertain this for sure, it is often necessary to simply approximate the distribution of $p(\theta|\mathcal{D})$ by a distribution peaked around a single value, θ^*.

While a natural choice for the location of θ^* at the mean of $p(\theta|\mathcal{D})$, a more convenient choice for θ^* is the maximum of $p(\theta|\mathcal{D})$. This is because finding θ^* then corresponds to an optimisation problem,

$$\theta^* = \text{argmax}_{\theta} p(\theta|\mathcal{D}) \ . \tag{5.20}$$

With all the machinery of optimisation at our disposal, such a problem is relatively simple to tackle (compared with finding the expected value of $p(\theta|\mathcal{D})$). Approximating $p(\theta|\mathcal{D})$ in this way is known as Maximum a posteriori estimation (MAP estimation).

To proceed with MAP estimation, we can now use Bayes' Theorem to express the MAP optimisation problem equivalently in terms of our model's predictions, which we can then calculate. The form is:

$$\theta^* = \text{argmax}_\theta \left[\mathcal{Z}\ \mathcal{L}(\theta; \mathcal{D}) p(\theta)\right] \ . \tag{5.21}$$

As usual when performing optimisations, it is more convenient to minimise sums rather than maximise products. Therefore, as before with the likelihood, we take logs and multiply by a minus sign to get:

$$\begin{aligned}
\theta^* &= \text{argmin}_\theta \ -\mathcal{Z}\ \mathcal{L}(\theta; \mathcal{D}) p(\theta) \ , && (5.22)\\
&= \text{argmin}_\theta - \log\left[\mathcal{Z}\ \mathcal{L}(\theta; \mathcal{D}) p(\theta)\right] \ , && (5.23)\\
&= \text{argmin}_\theta - \log\left[\mathcal{L}(\theta; \mathcal{D})\right] - \log\left[p(\theta)\right] - \log\left[\mathcal{Z}\right] \ , && (5.24)\\
&= \text{argmin}_\theta - \log\left[\mathcal{L}(\theta; \mathcal{D})\right] - \log\left[p(\theta)\right] \ , && (5.25)\\
&= \text{argmin}_\theta - \log\left[\mathcal{L}(\theta; \mathcal{D})\right] - \log\left[p(\theta)\right] \ , && (5.26)
\end{aligned}$$

where in the third line we drop the term $\log\left[\mathcal{Z}\right]$ because it is not a function of θ and is, therefore, irrelevant for determining the location of the minimum.

Note that the first term in this minimisation problem is just the negative-log-likelihood. In fact, if we choose a *flat-prior*, i.e. one that does not depend on θ corresponding to no prior belief about the values of θ, then MAP reduces to minimisation of the negative-log-likelihood, since the prior term is now θ-independent and therefore irrelevant. In other words, maximum likelihood estimation is contained in MAP as a special case, helping to justify a procedure that was, until now, rather ad hoc.

In fact, the form of the MAP problem above is exactly that of the regularised MLE problem, i.e. minimising not just the negative-log-likelihood, $L(\theta; \mathcal{D})$, but $J(\theta)$. Comparing with the above expression, we can see that the regularisation term, $R(\theta)$, contained in $J(\theta)$ corresponds to the negative-log-prior:

$$R(\theta) = -\log\left[p(\theta)\right] \ . \tag{5.27}$$

Therefore, we can think of this kind of regularisation scheme, where we simply add a term to our loss that is independent of the data, as encoding prior belief about the parameter values into our problem. Rather than simply trusting the data entirely to determine the value of θ^*, we constrain it so that only in the case of sufficient evidence (e.g. a lot of data) will we be convinced that the parameters take the MLE form. This helps to reduce overfitting, because we know that we cannot simply trust the data we use to fit the model alone to produce a model that generalises well, as the model will be overly specialised to the data when we do not have sufficient data to combat model complexity.

5.5 Weight Decay as a Gaussian Prior

The specific example of weight-decay (l2) regularisation for a Gaussian linear model can be shown as a special case of MAP estimation where a Gaussian prior for the parameters $\boldsymbol{\beta}$ is chosen.

We will proceed using the case of linear regression as an example, where $\theta = (\boldsymbol{\beta}, \sigma^2)$, and choose a prior of i.i.d Gaussians for the β_p weights (i.e. the parameters excluding β_0) along with an uninformative prior for σ^2 and β_0,

$$p(\theta) = A \prod_{p=1}^{P} \mathcal{N}\left(\beta_p | 0.\tau^2\right) \ , \tag{5.28}$$

Here the factor A is just the overall constant factor determined by normalisation of probabilities (or probability densities). Note we have made the iid assumption for the priors, and the priors for σ^2 and β_0 are absorbed into the definition of A, since they are flat.

The log-prior appearing in the MAP optimisation problem is then;

$$\log\left[p(\theta)\right] = \log[A] + \sum_{p=1}^{P} \log\left[\mathcal{N}\left(\beta_p|0,\tau^2\right)\right] \tag{5.29}$$

$$= \log[A] + \sum_{p=1}^{P} \log\left[\frac{1}{\sqrt{2\pi\tau^2}} \exp\left(-\frac{1}{2\tau^2}\beta_p^2\right)\right] \tag{5.30}$$

$$= \log[A] + \sum_{p=1}^{P} \log\left[\frac{1}{\sqrt{2\pi\tau^2}}\right] + \sum_{p=1}^{P}\left(-\frac{1}{2\tau^2}\beta_p^2\right) \tag{5.31}$$

$$= \log[A] + P\log\left[\frac{1}{\sqrt{2\pi\tau^2}}\right] - \frac{1}{2\tau^2}\sum_{p=1}^{P}\left(\beta_p^2\right) \tag{5.32}$$

$$= \log[A] + P\log\left[\frac{1}{\sqrt{2\pi\tau^2}}\right] - \frac{1}{2\tau^2}(\boldsymbol{\beta}^T\boldsymbol{\beta} - \beta_0^2) \tag{5.33}$$

In the MAP optimisation problem, the first two terms in this expression – which do not contain any parameters being optimised – will drop out;

$$\theta^* = \text{argmin}_\theta - \log[A] - \log\left[p\left(\mathcal{D}|\theta\right)\right] - P\log\left[\frac{1}{\sqrt{2\pi\tau^2}}\right] + \frac{1}{2\tau^2}(\boldsymbol{\beta}^T\boldsymbol{\beta} - \beta_0^2)\ , \tag{5.34}$$

$$= \text{argmin}_\theta - \log\left[p\left(\mathcal{D}|\theta\right)\right] + \frac{1}{2\tau^2}(\boldsymbol{\beta}^T\boldsymbol{\beta} - \beta_0^2)\ . \tag{5.35}$$

We can see that the prior is just the l2 regularisation term for the $\boldsymbol{\beta}$ parameters. This acts to reduce the overall size of the parameters, thus restricting the effective model complexity (e.g. by avoiding subtle cancellations of terms with high degree polynomial models).

Note that the l2 term corresponding to our choice of prior is multiplied by the inverse of the prior (hyper)parameter, τ^2. The larger τ^2 is the wider the prior Gaussians, i.e., the weaker the prior. This corresponds to a smaller coefficient for the l2 regularisation, making it less important in the optimisation. Ultimately, an "infinitely wide" (and therefore uninformative) Gaussian, $\tau^2 \to \infty$, would remove the regularisation term, and we would get back MLE. If the Gaussian is very thin, $\tau^2 \to 0$, then we have very strong prior belief that the parameters should be close to zero and, correspondingly, the regularisation is very strong.

Chapter 6

Computational Graphs and Neural Networks

6.1 Computational Graphs for Predictions

Computational graphs provide a way to visualise and define functions. For example, one might wish to visualise the computation needed to go from an example input, $\mathbf{x}$, to an output prediction, $\hat{y}$, for the linear model, $\hat{y} = \boldsymbol{\beta} \cdot \mathbf{x}$.

A computational graph consists of nodes, representing the variables, and directed edges, representing the flow of information in the computation. Taking the example of a linear model with $P = 3$, $K = 1$ so that $\hat{y} = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$, the corresponding computational graph would have five nodes for the five variables, and edges running from each x_p node into the $\hat{y}$, since the computation proceeds by taking the initial input variables and combining them to compute $\hat{y}$. The corresponding graph is shown in Fig. 6.1. Note that the order of computations flows from left to right, so that variables appearing to the right are evaluated later than those on the left, on whose values they may depend. Variables lying on the same vertical position can then be evaluated independently (in parallel) as they do not require any information about each other in order to be computed.

6.2 Computational Graphs for Basis Expansion (Feature Construction)

In the same way that predictions can be expressed as computational graphs, so can basis expansions. Recall that basis expansions provide a way of preprocessing the original inputs we are given into new inputs that can be fed into our models. For example, if we are given a single example input x, we can pre-process this with a polynomial basis expansion, e.g. by computing x^2, x^3 and so on. We can then feed these new values into our models as if they were usual inputs, thus allowing us to model more complicated relationships with respect to the original inputs.

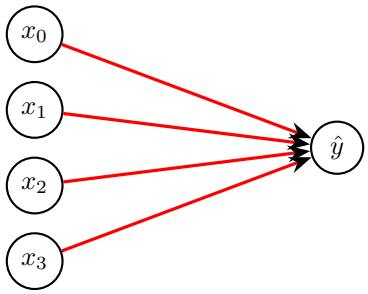

Figure 6.1: Computational Graph for a linear model

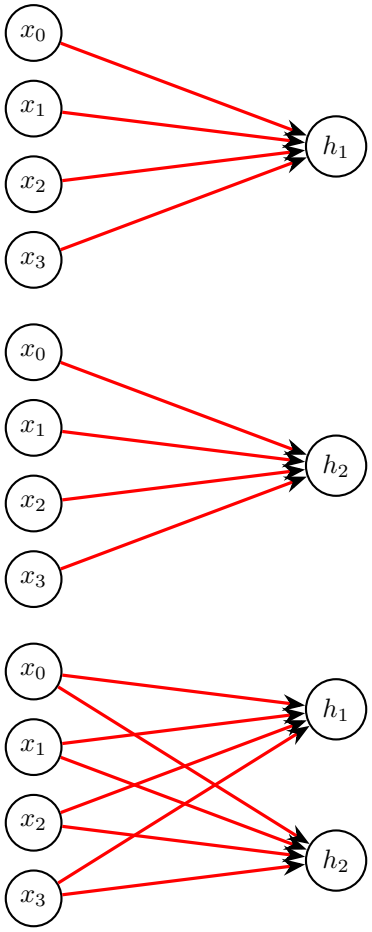

Figure 6.2: Computational graphs for the construction of two features from inputs.

To separate the concepts of the original inputs from those we have processed, we can use the term "inputs" to describe only the original data we are given, and "features" to describe the data after processing. Therefore, a basis expansion is a process of creating "features" from "inputs". In other words, it is a map from the input example $\mathbf{x}$ to the corresponding feature $\mathbf{h}$, where $\mathbf{h}$ is a vector of the features that we construct. One example, where again $P = 3$, would be to construct two features as;

$$h_1(\mathbf{x}) = \log(x_0 + x_1 + x_2 + x_3) \ , \tag{6.1}$$
$$h_2(\mathbf{x}) = \exp(x_0 x_1) + x_2 + x_3 \ . \tag{6.2}$$

Arranged in a vector, we then have;

$$\mathbf{h} = \begin{pmatrix} \log(x_0 + x_1 + x_2 + x_3) \\ \exp(x_0 x_1) + x_2 + x_3 \end{pmatrix} \ . \tag{6.3}$$

In terms of a computational graph, we can express the evaluation of features h_1 and h_2 separately, as shown in the first two graphs of Fig. 6.2. However, it is also possible to express both in a single graph, which represents how the computation of the whole feature vector, $\mathbf{h}$ proceeds from the input vector $\mathbf{x}$. The reason that this works is that h_1 and h_2 are computed independently of all the inputs, so they can be evaluated in parallel, and are thus placed on the same vertical line on the computational graph.

6.3 Computational Graphs for Feature Construction and Prediction

Having seen how we can express both feature construction and prediction separately as computational graphs, we can also combine them into a single graph. For example, we might wish to first process

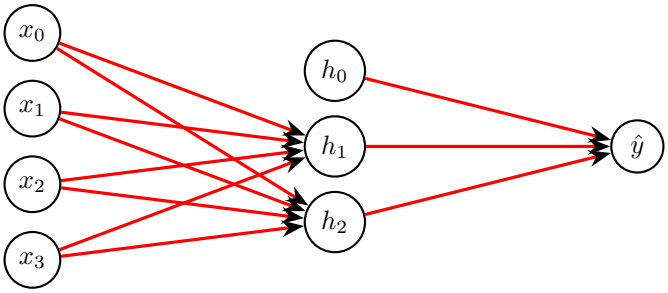

Figure 6.3: Computational graph for a linear model including a basis expansion

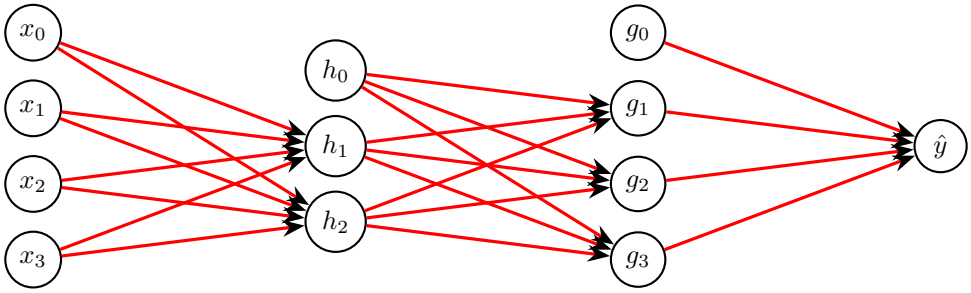

Figure 6.4: Computational Graph for a linear model with features constructed by two sequential operations, $\mathbf{x} \to \mathbf{h}$ and $\mathbf{h} \to \mathbf{g}$.

some input into features as $\mathbf{x} \to \mathbf{h}$, and then use the features for prediction in a linear model as $\hat{y} = \beta_0 h_0 + \beta_1 h_1 + \beta_2 h_2$, where $h_0 = 1$ by definition so that we can include an offset β_0 in our predictions. The corresponding computational graph representing the whole process $\mathbf{x} \to \hat{y}$ is shown in Fig. 6.3. Using the feature example from the previous section, this graph represents the function;

$$\hat{y} = \beta_0 + \beta_1 \log(x_0 + x_1 + x_2) + \beta_2 \exp(x_0 x_1) + \beta_2 x_2 \ . \tag{6.4}$$

Note that in the computation graph there is no connection into the node of h_0. This is because $h_0 = 1$ by definition and therefore requires no prior information to compute.

6.4 Layer-Unit Notation for Neural Networks

By placing both feature construction and prediction together in a single graph, we can see that it is possible to easily visualise even highly complex models that may contain many processing steps. For example, we may wish to perform more than one processing step on our inputs. First, we process the inputs $\mathbf{x}$ to create the features $\mathbf{h}$. Then we can take those features and process a new set of features $\mathbf{g}$. For instance, we might choose to process three new features as;

$$g_1 = h_0 + h_1 + h_2 \ , \tag{6.5}$$

$$g_2 = (h_0 + h_1 + h_2)^2 \ , \tag{6.6}$$

$$g_3 = (h_0 + h_1 + h_2)^3 \ . \tag{6.7}$$

In this case, writing an explicit expression for $\hat{y}$ in terms of the input $\mathbf{x}$ is rather complicated. However, the corresponding computational graph can be easily drawn as in Fig. 6.4

More generally, we can now consider defining a computational graph corresponding to a function in the following way. First, we choose the number of "layers" in the graph, $\mathbb{L}$, which we label as $l = 0, 1, ..., \mathbb{L}-1$. The zeroth layer corresponds to the inputs, $\mathbf{x}$, while the $(\mathbb{L}-1)$th layer corresponds to the outputs $\hat{y}$. The intermediate layers correspond to processing of the inputs, i.e. feature construction/basis expansions. The variables in each layer, corresponding to nodes that we will also call "units", can be arranged in a vector denoted as $\mathbf{h}^{[l]}$. Therefore, $\mathbf{h}^{[0]} = \mathbf{x}$ and $\mathbf{h}^{[\mathbb{L}-1]} = \hat{y}$. The components of these vectors refer to the individual units in the layer and can be denoted as $h_p^{[l]}$, such that $h_p^{[0]} = x_p$. The number of units in each layer will be denoted $P_l + 1$, with the "+1" coming from the inclusion of the usual zeroth component $h_0^{[l]} = 1$. Therefore, the dimension of the vector $\mathbf{h}^{[l]}$ is $P_l + 1$.

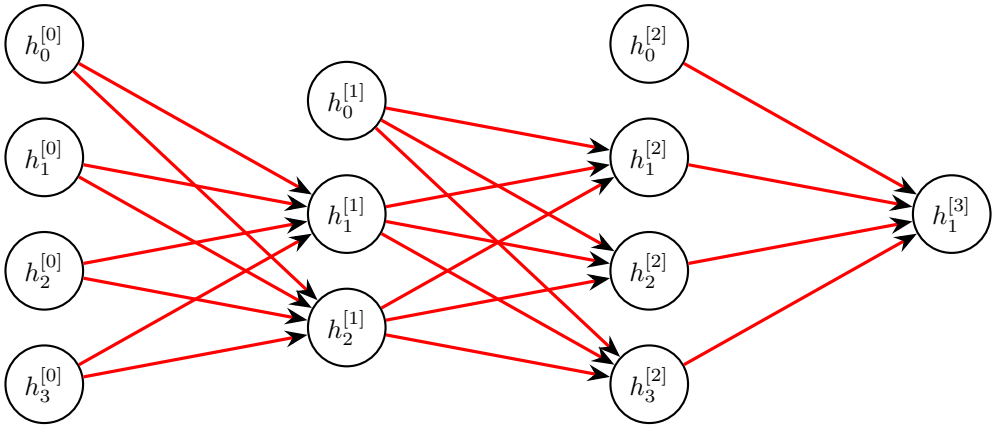

Figure 6.5: Computational Graph for a linear model with two features as in Fig. 6.4, written in the layer-unit notation.

The computational graph from Fig. 6.4 is rewritten in this new notation in Fig. 6.5. While this notation provides a flexible way to define computation graphs, with the possibility of having as many layers and as many units as we like, to complete the specification of the function we must now choose the functional form of each $\mathbf{h}^{[l]}$. Note that, by assumption, $\mathbf{h}^{[l+1]}$ is constructed using only the variables from the previous layer, $\mathbf{h}^{[l]}$. Therefore, to specify the overall function represented by the computational graph, we must now pick each function $\mathbf{h}^{[l]} \to \mathbf{h}^{[l+1]}$, as we had done previously.

6.5 Parametrised Basis Expansions: Activation Functions and Linear Combinations

While previously we had chosen the functional dependence of the variables in layer $l+1$ in terms of those of l by hand (e.g. by specifying the functions h_1, h_2 in terms of x_0, x_1 and x_2 explicitly) we can also consider only defining these functions up to some parameters. This is an important concept, as it means we will be able to automatically determine the best functions for the task in hand. In other words, just like we determined the best way to make predictions from our inputs by choosing the parameters of a parametric model, we will now be able to choose the best way to process our input examples by choosing parameters. Therefore, while previously we have been performing feature construction/basis expansions by hand, we will now begin to automatically determine the best features/basis expansion for the task in hand. This is a key concept in the construction of neural networks, and one that allows them to deal with complicated raw input data such as images, that would otherwise require a large amount of manual processing before predictions (such as to which class an image belongs) could be made.

A typical parametrisation for the units of a neural network is,

$$h_p^{[l+1]} = a_p^{[l+1]} \left(\boldsymbol{\beta}_p^{[l+1]} \cdot \mathbf{h}^{[l]} \right) \ . \tag{6.8}$$

What this expression means is that, for each unit in layer $l+1$, we first take the vector of units from the previous layer, $\mathbf{h}^{[l]}$, and compute a linear combination of these with respect to the parameter vector $\boldsymbol{\beta}_p^{[l+1]}$.

Note that this parameter vector has two labels, $[l+1]$ and p. The first of these refers to the layer. The second of these refers to the unit in that layer. Therefore, every unit in every layer is associated to its own set of parameters, except for the zeroth layer than has no inputs. After combining the previous layer's units linearly, so that only a single number remains, we apply a function $a_p^{[l+1]}$. This is typically referred to as the activation function.

Like the parameter vector, the activation function is labelled by both the layer and the unit, as each unit could potentially have a different activation function. The reason for including an activation function is to allow non-linear functions of the previous layer. For instance, the activation function could be a polynomial, though typically it is hyperbolic-tan (tanh) or a rectifier function, which have been found to work well in practice.

As an explicit example, let us take the four-layer computational graph depicted in Fig. 6.5. The units of the zeroth layer on the graph are just the inputs,

$$\mathbf{h}^{[0]} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \ . \tag{6.9}$$

The first layer then has three units total ($P_1 = 2$). If we choose all the activation functions in this layer to be tanh for illustration, then the variables in this layer can be expressed in vector notation as:

$$\mathbf{h}^{[1]} = \begin{pmatrix} h_0^{[1]} \\ h_1^{[1]} \\ h_2^{[1]} \end{pmatrix} = \begin{pmatrix} 1 \\ a_1^{[1]} \left(\boldsymbol{\beta}_1^{[1]} \cdot \mathbf{h}^{[0]} \right) \\ a_2^{[1]} \left(\boldsymbol{\beta}_2^{[1]} \cdot \mathbf{h}^{[0]} \right) \end{pmatrix} = \begin{pmatrix} 1 \\ a_1^{[1]} \left(\boldsymbol{\beta}_1^{[1]} \cdot \mathbf{x} \right) \\ a_2^{[1]} \left(\boldsymbol{\beta}_2^{[1]} \cdot \mathbf{x} \right) \end{pmatrix} = \begin{pmatrix} 1 \\ \tanh \left(\boldsymbol{\beta}_1^{[1]} \cdot \mathbf{x} \right) \\ \tanh \left(\boldsymbol{\beta}_2^{[1]} \cdot \mathbf{x} \right) \end{pmatrix} \ . \tag{6.10}$$

The next layer 2 can be expressed similarly in terms of the previous layers variables. Again choosing all the activation functions to be tanh for this layer as well, we have:

$$\mathbf{h}^{[2]} = \begin{pmatrix} h_0^{[2]} \\ h_1^{[2]} \\ h_2^{[2]} \\ h_3^{[2]} \end{pmatrix} = \begin{pmatrix} 1 \\ a_1^{[2]} \left(\boldsymbol{\beta}_1^{[2]} \cdot \mathbf{h}^{[1]} \right) \\ a_2^{[2]} \left(\boldsymbol{\beta}_2^{[2]} \cdot \mathbf{h}^{[1]} \right) \\ a_3^{[2]} \left(\boldsymbol{\beta}_3^{[2]} \cdot \mathbf{h}^{[1]} \right) \end{pmatrix} = \begin{pmatrix} 1 \\ \tanh \left(\boldsymbol{\beta}_1^{[2]} \cdot \mathbf{h}^{[1]} \right) \\ \tanh \left(\boldsymbol{\beta}_2^{[2]} \cdot \mathbf{h}^{[1]} \right) \\ \tanh \left(\boldsymbol{\beta}_2^{[2]} \cdot \mathbf{h}^{[1]} \right) \end{pmatrix} \ . \tag{6.11}$$

Finally, for the last layer, which computes the prediction, we could choose a single variable prediction, ($K = 1$), and a sigmoid activation function, S, as was the case for logistic regression. The final prediction variable then reads,

$$\hat{p} = S \left(\boldsymbol{\beta}^{[3]} \cdot \mathbf{h}^{[2]} \right) \ , \tag{6.12}$$

where we have dropped the subscript label for this layer's parameter vectors, because there is only one unit.

Part II

Unsupervised Learning

Chapter 7

Basics of Unsupervised Learning

7.1 Difference between Supervised Learning (SL) and Unsupervised Learning (USL)

Before going into the details about USL look at the grid shown in Fig. 7.1. This schematic illustrates the difference between the tasks and approaches taken by SL and USL. This is described in the following.

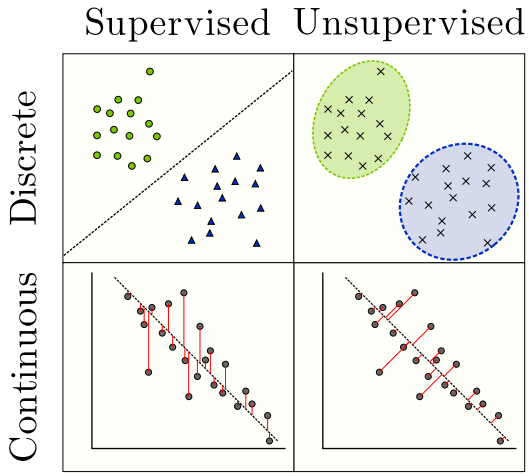

Figure 7.1: Differences between SL and USL. The top left sketches a SL classification task on a data set in which the classification is in terms of labels that take discrete values. The bottom left is a SL regression task corresponding to finding the best fit y (or prediction) to data x both of which (or at least the target) take continuous values (NB that the error indicated by the red drop lines is given by the distance from the actual y to the prediction $\hat{y}$). The top right is an USL clustering task, where data is aggregated into previously unknown discrete set of clusters (the task is to find such clusters). In the bottom right each axis corresponds to a component of each datum, say (x_1, x_2), and the USL task is to find the best curve that describes the continuous "manifold" on which the data lives (NB that the error in this case is the distance from the data to the line).

7.1.1 Discrete target

Here "discrete" refers to the discrete nature of labels or "targets". In the SL case these are known and the categorization or classification of the input is learned from a training set of labelled data, see Fig.1 top left. In the USL case there are no labels (or are not known in advance), and the input data is clustered together to view its structure, see Fig.1 top right. The USL task is to find these unknown clusters from the training data set.

7.1.2 Continuous target

Here "continuous" refers to the fact that the target takes continuous values. In the SL case this is simple regression as we have seen in previous lectures, finding the best estimate or prediction $\hat{y}(x)$. The USL task is somewhat more involved, or more abstract. In the example shown that data is two-dimensional, with each point indicating the location of each datum in terms of its two components (x_1, x_2). The USL is to find a curve on the plane that minimises the distance between points (imagine having to design a road that goes through a series of houses scattered in the countryside, only from data from a subset of the total set of houses it has to serve - i.e. it should both cater for the seen data and generalise well to the unseen data; more on all of this below). In this lecture we will go into detail of such USL tasks by exploring Principal Component Analysis method.

7.2 Connecting the main concepts of SL to those of USL

While different in the detail, many of the concepts of USL are related to those of Supervised Learning. Therefore, it makes sense to phrase the problem of USL the language we have seen already in the context of SL. Let us consider specifically what experience (E), task (T) and performance measure (P) are in USL.

7.2.1 Experience (E)

The main distinction between USL and SL is the Experience. The data for SL is structured into input and target quantities (e.g. the input data and their labels), where in contrast the data for USL only consists of input quantities and no labels. Conceptually this might seem a bit strange, since data with a target comes with clear tasks (i.e., match the label to the input), while without the target information the task to be performed is perhaps less clear. Nevertheless, as we will see next, tasks can be clearly defined in the USL case as well.

7.2.2 Task (T)

The task of a SL problem, such as the examples we have seen previously, is one of prediction. From the data provided, E, learn a relationship between the input and the target such that the model produced can be used to predict the target of an unseen piece of data. For the Arctic Ice example this might be looking at a future date to estimate the level of Ice at that future time. The task here is clear, well-defined and easily understood.

USL does not provide such an obvious structure to its tasks. Since there is no target in USL, the task cannot be one of prediction in the same sense. However, trying to analyse the input data to look for unknown structure is more plausible. Let us see what we mean by "structure in the data".

Let us consider a simple categorization problem, for example that of images of cats and dogs. The experience E is various pictures of cats and dogs: the input is the picture itself and the target is the label "cat" or "dog". Now imagine if we had never seen a cat or a dog before and someone gave us all of these pictures and asked us "Sort these pictures out for me!". How would we do it? Well, we might start by trying to find a (as yet unknown) feature that is a main distinguishing factor between some images. For example, some faces are longer and some of them are rounder, some have more prominent whiskers than the others, and so on. We would keep doing this until we have found a way to sort out all of the pictures into piles. When the person who gave us this task comes back, they can then look through the piles and see how we sorted them. If this person has seen cats and dogs before they should - hopefully - find that we have made a pile of cats and a pile of dogs. This is an USL task, find previously unknown general characteristics in data sets.

Of course in that example, however unrealistic, there are a few things we assumed. First is the feature extraction. How did we know that the shape of the face was important? And why did we ignore the fact that they all had fur? Feature extraction is an important part of USL, since often the data being used will contain large amounts of redundant information. We ignored the fact that all the animals in the pictures had fur because it did not help us to distinguish the data. However, we noticed a larger amount of distinction between the data when we looked at face shape. One of the ways we will be implementing this feature extraction will be Principal Component Analysis, which orders the features of the data allowing us to simplify the highly dimensional problem into a few important features.

We also assumed there were only two categories. When we were initially given the task we did not know how many categories there were, so we could have ended up with more than two piles. Knowing how to collect data is difficult, and there are a lot of methods that focus on grouping data. This is a major part of USL, known as clustering. The task of clustering is to take the data and cluster it, usually into a predetermined number of classes. But the number of classes is also a parameter that needs to be learned in order to analyse the data properly.

7.2.3 Performance Measure (P)

The final consideration is the performance measure. The definition of a performance measure is slightly more abstract for an USL task than in the SL case. In SL, it could be a simple measure of the error of how close to the target the prediction is (e.g. Mean Squared Error). In USL, with no target, the performance is measured by other means. For example, if we are trying to find the most important feature from our data set we will look for the feature that provides the largest variation among the data. While this might not be obvious at first, a little thought reveals that by increasing this variation we are increasing the distinctions between the data elements. Since our task is to be able to distinguish the data as easily as possible, this is a good performance measure of how well we are doing in our USL task.

Chapter 8

Principal Component Analysis

8.1 Dimensional Reduction

Dimensional reduction refers to methods that allow to describe approximately high dimensional data (i.e. where each datum is characterised by the value of a large number of variables, and thus corresponds to a point in a high dimensional space) in terms of a much smaller number of relevant features.

This is very common in physical sciences. Think for example of a material system (e.g. a liquid) whose state is fully described in terms of the positions of a large number of microscopic components. Such "microstate" is a point in a large dimensional configuration space. However, the set off all microstates of the system at certain conditions (say at some fixed temperature and pressure) - i.e. the data - can be described in terms of a small number of collective variables (say the energy, or the density in different regions of space). In this way we construct approximate descriptions which are tractable due to the small number of relevant "features".

Let's consider an example in terms of a data set that we will be using below. Consider the case of American cities in the 1960s. We can describe each city by a large number of variables which are easily measurable. Let's choose these to be average household size, how much rainfall it experiences, how educated the population is, and so on. We can compose a whole list of different characteristics of each city we look at. But many of them will measure related properties and so will be redundant. If so, we should be able to summarize each city with fewer - and more collective - features. This is the primary goal of dimensional reduction.

We now consider the first basic method of USL for dimensional reduction, in order to uncover unknown features in data, called Principal Component Analysis or PCA.

8.2 The PCA method

The purpose of PCA is to identify the most important features that can be used to identify relationships between unknown groups within the data. This is not done by selecting some variables and discarding others. Instead, the goal of PCA is to find linear combinations of the initial variables into a small number of features that summarise data well. These features are called "components" for reasons that will become clear below. PCA finds the best possible new features, the ones that summarise the data as well as possible when considering all possible linear combinations of the original variables.

8.3 The process of PCA

Let's work through how one would go about calculating the optimal components for a particular set of data. Firstly we outline our data set taken from the pollution data for cities available from the module moodle page [and also from here http://keel.es/datasets.php].

The data is for $N = 60$ cities, with each city characterised by $D = 16$ variables (or attributes, or "raw" unprocessed "features"). From now, we will call these variables just features (since the emerging

relevant features will be called "components"). Each datum $\vec{X}^{(j)}$ is a vector of $D = 16$ components, $\vec{X}^{(j)} = (X_1^{(j)}, \cdots, X_D^{(j)})$. In the dataset there are a total of N of these vectors, so $j = 1, \ldots, N$, and N is the size of the data set. As explained in previous weeks, we combine the data into a $N \times D = 60 \times 16$ matrix (the data matrix), where rows indicate cities and columns their corresponding features, which for the example we are considering reads

$$\mathbf{X} = \begin{pmatrix} \vec{X}^{(1)} \\ \vdots \\ \vec{X}^{(N)} \end{pmatrix} = \begin{pmatrix} 36 & 27 & ... & 921.87 \\ 35 & 23 & ... & 997.875 \\ ... & ... & ... & ... \\ 38 & 28 & ... & 954.442 \end{pmatrix} . \tag{8.1}$$

Recall the notation of the previous weeks: the elements of the design matrix $\mathbf{X}$ are $X_p^{(j)}$ corresponding to (raw) feature p for city (datum) j.

This is a high dimensional problem. With no knowledge on how the data is grouped, i.e. how the variables or raw features relate to each other, it is difficult to visualise to see how the data is clustered and find relations from direct inspection. Via PCA, we want to select a few of the most important components (linear combinations of the variables) that accurately describe the data and can be plotted in lower dimensions to make the analysis more transparent.

From the data, $\mathbf{X}$, we first calculate the vector of means (which is 16 dimensional) of each of the individual variable or raw feature,

$$\bar{X} = N^{-1} \sum_{j=1}^{N} \vec{X}^{(j)} = \begin{pmatrix} 37.3667 & 33.9833 & ... & 940.358 \end{pmatrix} . \tag{8.2}$$

The aim is to remove the mean from the data, as this simply corresponds to an overall translation in the space where the data lives and carries no information about the actual relationships between the variables.

The second step is to calculate the *covariance matrix* of the whole data set,

$$\mathbf{\Sigma} = \frac{1}{N-1} \sum_{j=1}^{N} \left(\vec{X}^{(j)} - \bar{X} \right)^T \cdot \left(\vec{X}^{(j)} - \bar{X} \right) . \tag{8.3}$$

Note that in our convention $\vec{X}^{(j)}$ and $\bar{X}$ are row vectors, cf. Eqs. (8.1,8.2). The equation above is then the sum of the dot product of column vectors of dimension D and row vectors of dimension D. This makes $\mathbf{\Sigma}$ a $D \times D$ square matrix.[1]

Written in terms of its elements, the covariance matrix is

$$\Sigma_{p,p'} = \frac{1}{N-1} \sum_{j=1}^{N} \left(X_p^{(j)} - \bar{X}_p \right) \left(X_{p'}^{(j)} - \bar{X}_{p'} \right) , \tag{8.4}$$

where $\bar{X}_p$ are the elements of the vector of means Eq.(8.2).

The covariance matrix is such that its diagonals are the variance over the data for each feature, and the off-diagonals encode the cross-correlations given the data between pairs of different features. For the case we are considering it looks explicitly,

$$\mathbf{\Sigma} = \begin{pmatrix} 99.6938 & 9.36215 & ... & 316.455 \\ 9.36215 & 103.406 & ... & -18.9897 \\ ... & ... & ... & ... \\ 316.455 & -18.9897 & ... & 3869.62 \end{pmatrix} , \tag{8.5}$$

We see that it is symmetric around the diagonal, as expected from Eq.(8.3), and has dimensions given by the number of features, i.e., it is a $D \times D$ matrix, 16×16 for our case.

[1] Our convention is that vectors written as $\vec{V}$ are row vectors, while those written as transposed, $\vec{V}^T$, are column vectors. This comes from the definition of the data matrix in Eq. (8.1) and from previous lectures. Sometimes we omit indicating whether it is the vector or its transpose if the context of the equation makes it obvious which one it is. For example, the expression $\mathbf{\Sigma} \cdot \vec{V}^T$ can equally be written $\mathbf{\Sigma} \cdot \vec{V}$, as the only way to multiply a $D \times D$ matrix by a D dimensional vector on the right is if the vector is written as a column.

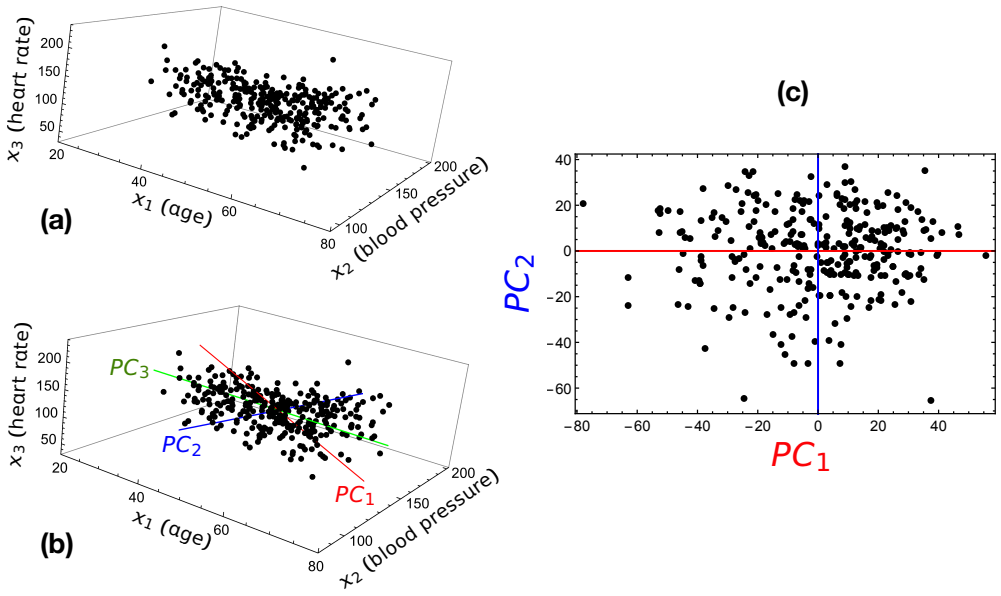

Figure 8.1: PCA for heart disease data [from https://www.kaggle.com/ronitf/heart-disease-uci, see also moodle page]. The data set is three-dimensional, $D = 3$, with the raw features corresponding to x_1 =age, x_2 =blood pressure at rest, and x_3 =maximum heart rate. Panel (a) plots all the data. Panel (b) shows the three PCs. Panel (c) plots all the data points projected onto the plane defined by PC_1 and PC_2.

The next step is to calculate the eigenvalues and eigenvectors of $\mathbf{\Sigma}$. That is, we wish to compute unit vectors $\vec{v}$ that obey

$$\mathbf{\Sigma} \cdot \vec{v} = \lambda \vec{v} \ , \tag{8.6}$$

where λ is the eigenvalue of eigenvector $\vec{v}$. (See footnote 1). The eigenvalues are obtained from the characteristic polynomial of the matrix,

$$\det(\mathbf{\Sigma} - \lambda \mathbf{I}) = 0 \ , \tag{8.7}$$

where $\mathbf{I}$ is the identity matrix. For each eigenvalue λ, the corresponding eigenvector is obtained by replacing λ into Eq.(8.6) and solving for $\vec{v}$. In the case of our covariance matrix the number of eigenpairs is equal to the number P of (raw) features of our data.

These eigenvectors are the components. To choose the *principal* component we simply look for the eigenpair with the largest eigenvalue. If we sort the eigenpairs by eigenvalues we then have a ranking of importance of all of our new components.

PCA is illustrated in Fig. 8.1.

8.4 How does PCA work?

The natural question to ask now is how does PCA actually achieve the task of selecting the most relevant combination of features to reduce dimensionality in the description of the data? To answer this question we should break up the task itself.

Firstly, in order to find the most important combination of the variables or raw features, it is intuitively clear that one should look for the combination that differs most strongly between the various independent samples. Consider the 60 cities in the example we are focused on above. Imagine that you come up with a feature that is nearly the same for most of the cities. This would not be very useful to distinguish the data. You might want to find out what makes one city different from another or to identify if there are certain correlated features for every city in order to try and improve some aspect of one particular city. In either case, a variable that encodes little to no variation will not be of much use when you are trying to find the most important features of a city. This would certainly be a poor way to summarise the

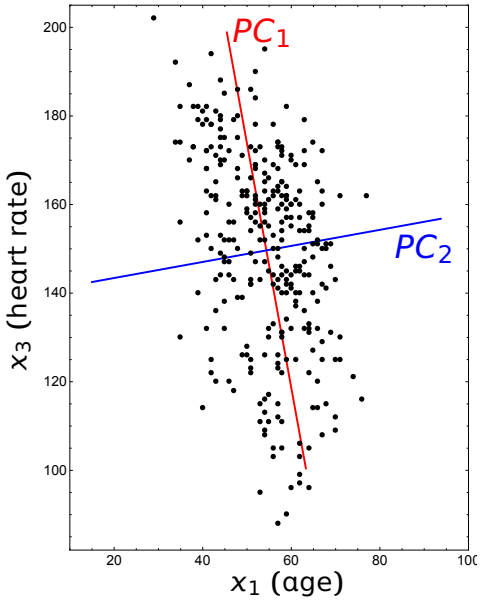

Figure 8.2: PCA for heart disease data [from https://www.kaggle.com/ronitf/heart-disease-uci, see also moodle page]. The data set is two-dimensional in this case, $D = 2$, with the raw features corresponding to x_1 =age, x_2 =maximum heart rate. The plot shows the data and the two PCs.

description of the data (i.e., to reduce dimensionality). Instead, PCA looks for combination of features - the components - that show as much variation across the data set as possible.

Secondly, a sightly more subtle task of PCA is to find components that would allow to predict some of the original features with the least amount of error. This in essence is exactly what you would do in a linear regression. Remember USL does not have the same data as SL, in that there are no labels to train on, and predict from, your data. But if, in the USL case, feature p of your data set has some missing values from some of your samples, you might want to use another feature of your data p' to try and predict what those missing values should be. In other words PCA tries to reduce the error of prediction when you use another feature to learn that prediction.

The reason we have outlined these two tasks of PCA is that they are actually the same. Minimising the error of prediction is equivalent to maximising the variance of the data for the component. Even though one of them we have already seen in the supervised case, it is a relevant description of this USL method.

8.5 Example of PCA

As an illustration we consider another example, that of heart disease data[2]. While the original data set has dimension fourteen, for illustration purposes we consider only three of raw features. This raw data with dimension $D = 3$ for each of the $N = 303$ data points is plotted in Fig. 8.1(a).

The PCs can be calculated as explained above. They are plotted together with the data in Fig. 8.1(b). The directions defined by the PCs go through the centre of the data, with PC_1 being the red direction, PC_2 the blue one and PC3 the green one. These are three orthogonal directions in the data space, with PC_1 describing that of maximum spread of points. This is more clearly seen by projecting all the data points on the plane defined by PC_1 and PC_2, as shown in Fig. 8.1(c). That is, we plot the set of two-dimensional vectors with components $(\vec{X}^{(j)} - \bar{X}) \cdot \mathrm{PC}_1$ and $(\vec{X}^{(j)} - \bar{X}) \cdot \mathrm{PC}_2$.

8.5.1 Maximising the variance

Consider the example of Fig. 8.2 which for simplicity has data with two raw features. The data is centred around the point $\bar{X}$ (located where the red and blue lines cross).

[2]From https://www.kaggle.com/datasets/redwankarimsony/heart-disease-data, see also PHYS4035 Moodle page.

We wish to find a line, passing through the mean such that the data when projected on this line is spread as much as possible. To simplify the description we shift the origin of our coordinate system to the location of the mean by subtracting the mean $\bar{X}$ from each data point. Let's define the vector for each pointed thus shifted as

$$\vec{x}^{(j)} = \vec{X}^{(j)} - \bar{X}\,. \tag{8.8}$$

The direction of the line is given by a unit vector $\vec{V}$ (which will be the red line in Fig.8.2). We can then project each data point (after subtraction of the mean) on the line. The variance of the projected data is given by

$$\sigma_V^2 = N^{-1}\sum_{j=1}^{N}\left(\vec{x}^{(j)}\cdot\vec{V}^T\right)^2\,. \tag{8.9}$$

The variance is a measure of how spread out points are along the direction defined by $\vec{V}$. Our first aim is to find a direction $\vec{V}$ that maximises σ_V^2.

8.5.2 Minimising the error

In SL (e.g. linear regression) we know one of the variables is dependent on the other(s), which we often phrase by saying that the label (or the "target") is dependent on the input. Since the goal in SL is to be able to predict the label, for each input we want to minimise the error on the label. That is, the input is given, and the error is in the estimation of the label for a given (i.e., errorless) input.

In the USL case this is no longer the case. There is no distinction between label and input, all the variables represent the data equally. This means we need to minimise the error in the prediction with respect to all of the variables or raw features. In the PCA case we are discussing, this is equivalent to saying that the error is given by the distance from the data to the line that describes the optimal direction with which to describe the data, the PC_1.

For each point $\vec{x}^{(j)}$ we can define a unit vector $\vec{B}_j$ orthogonal to $\vec{A}$ in the direction from the line defined by $\vec{V}$ to the point. The distance squared of the point to the line is then given by

$$\left(\vec{x}^{(j)}\cdot\vec{B}_j^T\right)^2\,. \tag{8.10}$$

Summing over all the points we get a measure of the overall distance (squared) of the data to the line defined by $\vec{V}$,

$$\sigma_B^2 = N^{-1}\sum_{j=1}^{N}\left(\vec{x}^{(j)}\cdot\vec{B}_j^T\right)^2\,. \tag{8.11}$$

This quantifies the error made by describing the data via a linear manifold. Our second aim is to minimise this error.

8.5.3 Why they are equivalent

We have now described two quantities that we are trying to maximise (the spread along the line defined by $\vec{V}$) and minimise (the distance to the line), Eqs.(8.9) and (8.11), respectively. It is easy to see that these two task are actually the same.

Each point can be written as the projection on $\vec{V}$ in the $\vec{V}$ direction plus the projection on $\vec{B}_j$ in the $\vec{B}_j$ direction,

$$\vec{x}^{(j)} = (\vec{x}^{(j)}\cdot\vec{V}^T)\vec{V} + (\vec{x}^{(j)}\cdot\vec{B}_j^T)\vec{B}_j \tag{8.12}$$

Then for each point the distance squared to the origin is given by

$$\vec{x}^{(j)}\cdot\vec{x}^{(j)} = (\vec{x}^{(j)}\cdot\vec{V}^T)^2 + (\vec{x}^{(j)}\cdot\vec{B}_j^T)^2 \tag{8.13}$$

If we now take the mean over all the data points we get that the average distance squared is related to Eqs.(8.9) and (8.11) as

$$\bar{d}^2 = N^{-1} \sum_{j=1}^{N} \vec{x}_j \cdot \vec{x}_j^T = \sigma_A^2 + \sigma_B^2 \ . \tag{8.14}$$

Since the l.h.s. is fixed, maximising the variance σ_A^2 is the same as minimising the error σ_B^2. This means that finding the direction A along which the data is most spread out automatically gives us the smallest error in the distance of the data points to the line defined by A.

Here we have analysed the case where the manifold is a line defined by a single vector (cf. a single PC). A simple generalisation of the discussion above applies if the manifold is a (hyper) plane defined by several PCs.

8.5.4 Why the largest eigenvector is the best choice

In the example above we considered the optimal direction given by unit vector $\vec{A}$ in the subspace span by only two of the features, and mentioned that the optimal choice came from PCA1. That is $\vec{A}$ above was a two-dimensional vector obtained by projecting PCA1 onto the (p, p') plane. The aim is to find a vector $\vec{V}$ which is optimal *for all features*. We prove now that indeed the largest eigenvector of the covariance matrix - i.e., PCA1 - is indeed that optimal direction in the full space of features.

Consider a d-dimensional vector $\hat{V}$. If we centre each data point around the average, $\vec{X}_j - \bar{X}$ for point j, then the projection of this shifted point on the direction of $\hat{V}$ is given by

$$\vec{V} \cdot (\vec{X}_j - \bar{X})^T \ . \tag{8.15}$$

To remove any unnecessary scale factor from $\vec{V}$ we request that it is a unit vector

$$||\vec{V}|| = \vec{V} \cdot \vec{V}^T = 1 \ . \tag{8.16}$$

We discussed why we would want to maximise the variance of our data along the direction of $\vec{V}$, but it might not yet be clear as to why this is related to the eigenvectors of the covariance matrix. The reason is the following. If we project the covariance matrix, Eq.(8.3), along the $\vec{V}$ direction we get

$$\vec{V} \cdot \mathbf{\Sigma} \cdot \vec{V}^T = \frac{1}{N-1} \sum_{j=1}^{N} \left[\vec{V} \cdot (\vec{X}_j - \bar{X})^T \right]^2 , \tag{8.17}$$

where we have used the fact that $\vec{V} \cdot (\vec{X}_j - \bar{X})^T = (\vec{X}_j - \bar{X}) \cdot \vec{V}^T$. The r.h.s. is (up to a factor) the variance of the points along the direction of $\hat{V}$

$$\sigma_V^2 = \frac{1}{N} \sum_{j=1}^{N} \left[\vec{V} \cdot (\vec{X}_j - \bar{X})^T \right]^2 = \frac{N-1}{N} \vec{V} \cdot \mathbf{\Sigma} \cdot \vec{V}^T \tag{8.18}$$

which quantifies the spread in the projections of the data on the line defined by $\hat{V}$ passing through the overall mean, precisely the object we are trying to maximise.

Our aim is to find a $\vec{V}$ that maximises the projected covariance, Eq.(8.17), subject to the restriction that it has unit norm, Eq.(8.16). As is standard in maximisation and minimisation problems, we account for the constraint via a Lagrange multiplier. That is, we want to maximise that quantity

$$L(\vec{V}) = \vec{V} \cdot \mathbf{\Sigma} \cdot \vec{V}^T - \lambda \left(\vec{V} \cdot \vec{V}^T - 1 \right) \ . \tag{8.19}$$

with respect to $\vec{V}$ and with the Lagrange multiplier λ being such that the solution satisfies Eq.(8.16). This is a quadratic "loss function" and the calculation is not hard.

Let us do it first using vector calculus. The extreme of the (scalar) function $L(\vec{V})$ is given when the gradient with respect to $\vec{V}$ vanishes,

$$\nabla_{\vec{V}} L(\vec{V}) = 0 \tag{8.20}$$

$$\Rightarrow \mathbf{\Sigma} \cdot \vec{V} - \lambda \vec{V} = 0 \tag{8.21}$$

$$\Rightarrow \mathbf{\Sigma} \cdot \vec{V} = \lambda \vec{V} \tag{8.22}$$

This shows that the solution to the minimisation problem is given by when $\vec{V}$ is an eigenvector of $\mathbf{\Sigma}$ with eigenvalue λ. But which eigenvector, there are d of them after all? We answer this by replacing the answer above into the variance Eq.(8.18)

$$\sigma_V^2 \propto \vec{V} \cdot \mathbf{\Sigma} \cdot \vec{V}^T = \vec{V} \cdot \left(\lambda \vec{V}^T\right) = \lambda \vec{V} \cdot \vec{V}^T = \lambda \tag{8.23}$$

where we first dropped the N factors (they do not enter the discussion as N is fixed), then used the last line of Eq.(8.22), and finally the normalisation of $\vec{V}$. What we see is that the variance is determined by the eigenvalue λ, and therefore we maximise it by choosing $\vec{V}$ to be the eigenvector of $\mathbf{\Sigma}$ with maximum eigenvalue, thus proving in general the PCA method.

Chapter 9

Clustering Analysis

The main theme of Unsupervised Learning (USL) is structure analysis. In most USL tasks there is little to no information about the underlying structure of the data you have been given. One of the main approaches we have for extracting structure from data is called clustering. Simply put, clustering tries to group data together using information about similarities and differences in the data.

9.1 Difference between Clustering and Classification

Clustering might sound similar to classifying the data, which is a Supervised Learning (SL) task. The key difference as with all SL vs USL tasks is the lack of a label or target. Where the classification process is trying to focus on features that can be used to sort the data into specific known categories, clustering tries to do so in the absence of specific known categories as constraints.

The task of clustering can be considered as trying to find the groupings such that we, as the analyst, can give the groups labels or categories once these groups are identified via USL. Clearly this can be a powerful tool depending on the complexity of the features, since it offers a way to sift through large amounts of data and sort it out for us automatically. We still have to interpret the clusters for them to be useful, but knowing what type of clustering was used to find the groups can help inform us about the clusters themselves.

9.2 Example of clustering

Figure 9.1 shows an example of clustering (for details see the USL Week 2 Question Sheet). The left panel shows a data set of $N = 11$ two-dimensional data points. Visual inspection of the data suggests that there might be three distinct subsets of the data points, which an adequate clustering method may be able to find. The right panels are the successive iterations of the K-means algorithm described below. See the figure caption for details.

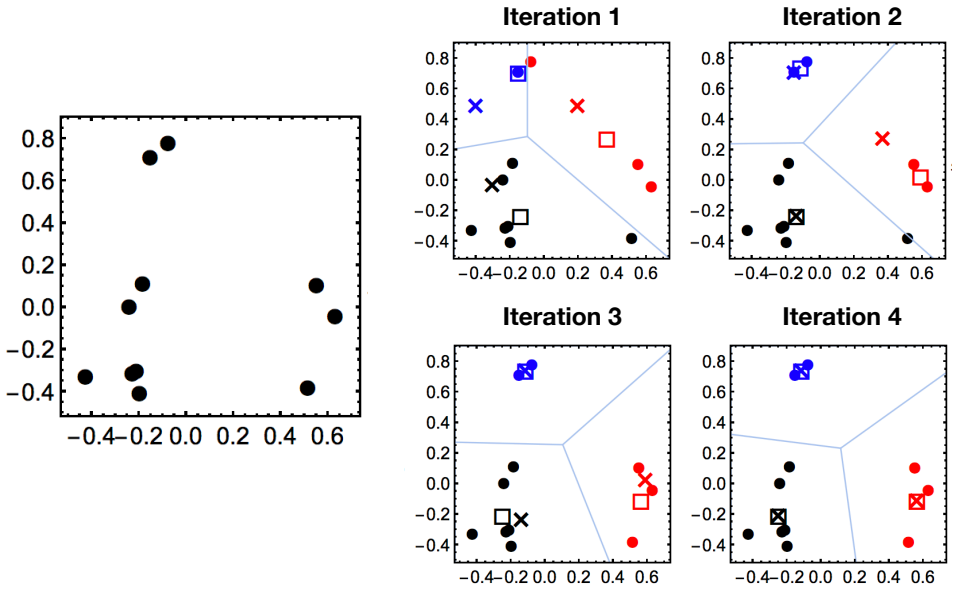

Figure 9.1: An example of an application of the K-means algorithm. The data set consists of $N = 11$ data points that live in the two-dimensional plane, see left panel. The right panels show successive iterations of K-means, with $K = 3$ clusters to be identified. In the first iteration, the centroids (indicated by crosses) are initialised in random locations. Each data point is then assigned to the closest centroid. The blue lines indicate the boundaries between the Voronoi cells that correspond to each centroid, so that each point belongs to one cluster (indicated by the colours). For each cluster, the mean position is computed, indicated by the open squares. In the next iteration the centroids are moved to the location of the means, and the process is repeated until convergence. In this example, after four iterations the centroids and the means coincide to a good approximation (indicated by the crosses lying on top of the square symbols). For details, see the USL Week 2 Question Sheet.

Chapter 10

Types of Clustering Algorithms

There are several approaches to clustering, and therefore different types of clustering algorithms. Let us outline some of the most commonly used ones.

10.1 K-means

This clustering algorithm computes the mean, or centroids, of the data associated to groups, and iteratively updates the position of the centroids, until the algorithm finds the optimal centroids. The key assumption is that the number K of clusters is known in advance, thus the name of the algorithm.

K-means can be thought of in a qualitative way as a bubble model. If the assumption is that there is a single cluster, $K = 1$, there will be a single bubble (which for simplicity we take to be spherical) that would encompass all data. The single centroid will correspond to the overall mean in feature space and the size of the bubble will be such that all data is enclosed. See Fig. 10.1 (left).

If instead we assume two groups or clusters, $K = 2$, then a second bubble is introduced. The two bubbles then "compete" for the data trying to group it in the optimal manner. Their overlap defines a line (or more generally a plane or hyperplane if considering $d > 2$) that separates the data. All the data is still described by both bubbles. See Fig. 10.1 (middle).

By increasing K new bubbles get added and consequently new (hyper)planes that separate the data. The location and size of the bubbles will change as they enclose their respective groups, and the separating planes prevent information of data from a one bubble to directly influence another one. See Fig. 10.1 (right).

The aim of the K-means method is to achieve this grouping in an optimal way.

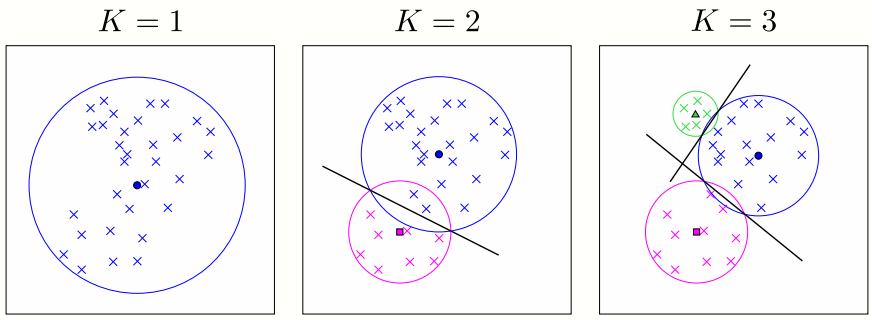

Figure 10.1: Shown here is a diagram of the output of K-means for made up data given a different number of clusters, K. The coloured clusters a basically circular domains, but when determining which cluster each data point belongs (for $K > 1$) to, we use the line defines as by the overlapping of the two circles.

10.2 Hierarchical Clustering

Hierarchical clustering is a different type of clustering that focuses on the distances between data points. This is distinct to K-means which focuses on distances to the centroids.

There are two approaches to hierarchical clustering. One is the top-down approach which starts from one cluster and splits it up successively. This is called Divisive Clustering. The second one is the bottom up approach which starts with each separate data point and tries to combine it with the closest data points into a cluster. This is called Agglomerative Clustering.

Since hierarchical clustering focuses on the distances between the data, it is clear that the choice of metric that describes that distance is important. A typical choice is the Euclidean distance, which is the distance between two points on a flat surface[1].

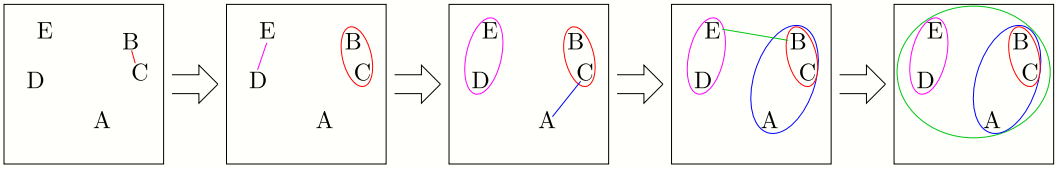

Figure 10.2: Iterations of Agglomerative Clustering for made up data.

Let's focus on Agglomerative Clustering as an example of how hierarchical clustering works. In Fig 10.2 we can see that with all the distances between data points calculated and ordered we can cluster data points into clusters. This process will get us $N-1$ clusters if there are N data points. Once the hierarchy of clusters has been constructed, we can then look at the clusters and analyse them. Clearly the largest cluster does not tell us anything, but the next level down would be the clusters of (A, B, C) and (E, D). These clusters are the first level of splitting of our data sufficiently, which might indicate and important (previously unknown) distinction between the two groups, leading to the analyser to find appropriate labels or categories.

10.3 Mixture of Gaussians

The third method we want to outline is the distribution model based clustering method. While the previous two approaches, K-means and hierarchical clusterings provide a sharp distinction between clusters (i.e., a data point belong to one cluster or another), distribution models give a probabilistic view of which cluster the data comes from.

The most common type of distribution model used is the Gaussian Mixture Model (GMM), where each cluster is represented by a Gaussian distribution. This is sketched in Fig. 10.3. The GMM is given by the following expression for d-dimensional data

$$p(\vec{x}) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(\vec{x}|\vec{\mu}_k, \mathbf{\Sigma}_k) \ . \tag{10.1}$$

The USL task it to learn the probability distribution $p(\vec{x})$ from the data, under the assumption that $p(\vec{x})$ is a linear combination of K (multidimensional) Gaussian distributions. In the expression above π_k is the weight of each Gaussian, with

$$\sum_{k=1}^{K} \pi_k = 1 \ , \tag{10.2}$$

and $\mathcal{N}(\vec{x}|\vec{\mu}_k, \mathbf{\Sigma}_k)$ represents a d-dimensional normal distribution (i.e., a Gaussian) with mean $\vec{\mu}_k$ and covariance matrix $\mathbf{\Sigma}_k$. Note that each individual Gaussian is normalised,

$$\int d^d x \, \mathcal{N}(\vec{x}|\vec{\mu}_k, \mathbf{\Sigma}_k) = 1 \ , \tag{10.3}$$

[1] In contrast, one could think of data that lives on the surface of a (hyper-)sphere, or other curved manifolds, in which case the distance between points would not be Euclidean.

and therefore we have that

$$\int d^d x \, p(\vec{x}) = 1 \ . \tag{10.4}$$

Note the following. The k-th Gaussian represents the k-the cluster. While each Gaussian will be concentrated on a cluster of point, since the Gaussian distribution has support for all values of $\vec{x}$, there is some contribution from all clusters to each data point. This means that the GMM gives non-zero probability for each point to belong to any cluster. This is the probabilistic nature of this approach to clustering.

A further important remark is the following. Given a data set, via a probabilistic model approach we can learn a best approximation for the probability $p(\vec{x})$ the data could have originated from. Since $p(\vec{x})$ in principle takes values in the whole possible domain of $\vec{x}$ (i.e., not only on the values of the data set) if we *sample* $p(\vec{x})$ we can *generate* new and distinct sample points compatible with the given data (think of generating images of new "fake" galaxies - on demand - from a set of actual astronomical observations). This is another very important application of USL, that of finding what are called *generative models*.

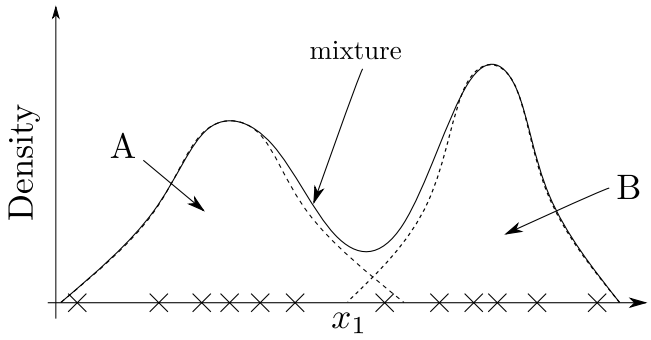

Figure 10.3: Sketch of a mixture of Gaussians approximation to the distribution $p(x_1)$ of the data (indicated by crosses). Here $d = 1$ (i.e., the data is one-dimensional). Shown here is a diagram of a mixture of Gaussian representing density the data along x_1. The dashed line represent the two different Gaussians A and B. The solid line is the mixture of the two.

Chapter 11

Clustering the city data

Having outlined a few different possible algorithms for the task at hand, we need to pick one to use. Let's study again the pollution data for cities considered in the USL Week 1 Lecture Notes [available from the module moodle page and also from here http://keel.es/datasets.php]. The first panel of Fig. 11.1 shows the data we want to analyse, corresponding to a two-dimensional reduction of the original $D = 16$ dimensional data by means of PCA: the panel plots the $N = 60$ data points along the two leading PCs. The goal is to analyse this data. Let us start by using K-means clustering.

11.1 K-mean algorithm

We briefly explained how K-means works in Sec.10.1. We now go into more detail on the algorithm.

Consider N data points, each represented by a d-dimensional vector $\vec{x}_i$, with $i = 1, \ldots, N$. The K-means algorithm proceeds as follows:

1. Decide on the number of clusters K.
2. Initialise K centroids by randomly choosing $\vec{\mu}_k$, with $k = 1, \ldots, K$.
3. Assign each data point $\vec{x}_i$ to a cluster S_k associated to the centroid for which the squared distance $\left\|\vec{x}_i - \vec{\mu}_k\right\|^2$ is the smallest. This defines the K clusters $\mathbf{S} = (S_1, \ldots, S_K)$, that is

$$S_k = \left\{\vec{x} \in \{\vec{x}_1, \ldots, \vec{x}_N\} : \left\|\vec{x} - \vec{\mu}_k\right\|^2 \leq \left\|\vec{x} - \vec{\mu}_{k'}\right\|^2 \ \forall\, 1 \leq k, k' \leq K\right\} . \tag{11.1}$$

4. Calculate the mean position of cluster S_k and assign it to the centroid $\vec{\mu}_k$, for all k,

$$\vec{\mu}_k = \frac{1}{|S_k|} \sum_{\vec{x} \in S_k} \vec{x} \ , \tag{11.2}$$

   where $|S_k|$ is the number of data points inside set S_k.

5. Repeat from 3 until convergence.

At this point it should be clear what we are actually trying to achieve. We are trying to reduce the square of the distance to the closest centroid for each point, and therefore find the centroids that define the optimal clusters by the following variance minimisation

$$\mathbf{S}^* = \arg\min_{\mathbf{S}} \sum_{k=1}^{K} \sum_{\vec{x} \in S_k} \left\|\vec{x} - \vec{\mu}_k\right\|^2 = \arg\min_{\mathbf{S}} \sum_{k=1}^{K} |S_k| \, \mathrm{var}_k(x) \ , \tag{11.3}$$

where we have defined $\mathrm{var}_k(x)$ the variance of the data within the group S_k. This means that in practice in K-means we are trying to find the grouping of the data with the least variation within each group.

For an example of how K-means converges, see Fig. 9.1.

11.1.1 How many clusters do we need?

This can be a difficult question to answer for data where the clusters are not evident, such as the city data. To be blunt this is a problem with clustering algorithms, since there is no sure fire way to guarantee you pick the correct number of clusters. There are a few methods, however, to try to have a best guess.

For this case the method we will use for selecting the number of clusters is called the *elbow method*. This simply requires us to go through the algorithm outlined in the previous section for a range of different values of K. We can see what this looks like on our data in Fig. 11.1.

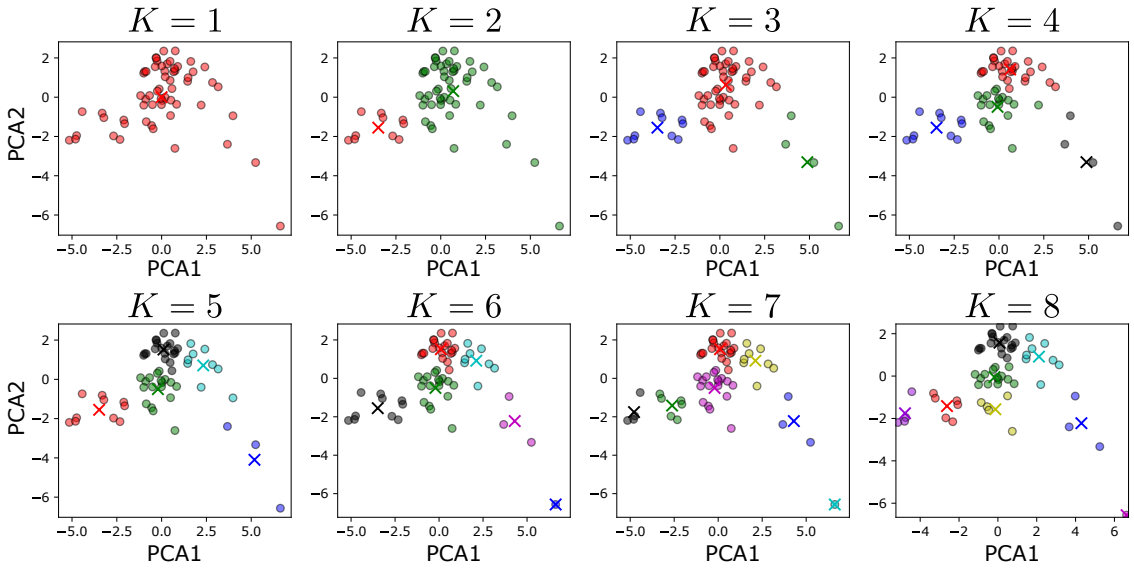

Figure 11.1: Each plot here shows the same data, from Fig. 9.1, but with a different number K of clusters trying to minimise the variance.

For each K we can calculate the total square distance from each data point to its associated centroid, as given by Eq. (11.3). Figure 11.2 shows that as the number of clusters increases the total variance decreases. This makes sense, since if we have $K = 60$ each cluster would have a variance of zero, since there would be only one data point per cluster coinciding with its centroid. However, such limit reveals nothing about the structure of the data set. This is an extreme case but shows there is some competition of usefulness between minimising the total variance and limiting the number of clusters.

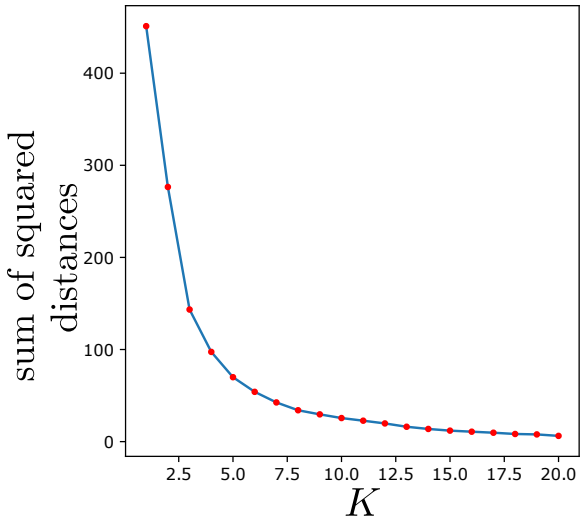

Figure 11.2: This plot shows how the total squared distance decreases as the number of clusters increases.

The elbow method is a way to select the value of K that requires us to look for the value of K where the

rate of decrease in total squared distance drops. Looking at Fig. 11.2, $K = 2$ drops significantly as does $K = 3$. After that however the drop is less significant, therefore further increase in K is less important than previous increases. That is, one is looking for an elbow (or turning point) in Fig. 11.2.

Another way to think of the number of clusters, would be to see it as under-fitting and over-fitting the data. If we have the number of clusters equal to the data points, any new data at that point would be very hard to fit to, since all the centroids would have been focused around a single point, this is over-fitting. Alternatively if we just had one cluster to fit our data, we would not have minimised our variance very well and thus would have under-fit our data. So we can see that even in USL we can have similar problems to those of SL.

11.1.2 How well does it perform?

Now let's look at the outcome of using this method on our city data. We settled on $K = 3$ clusters using the very simple and somewhat subjective elbow method. Looking at Fig. 11.3 we can see the result of the analysis.

It is worth making a few comments here. The initialisation of the centroids is random. Therefore, to make sure we are not experiencing any bias from the initial conditions it is good to repeat the procedure starting from multiple different random initial conditions for the centroids. A robust result should be mostly independent on the initial choice, i.e., a genuine property of the data set.

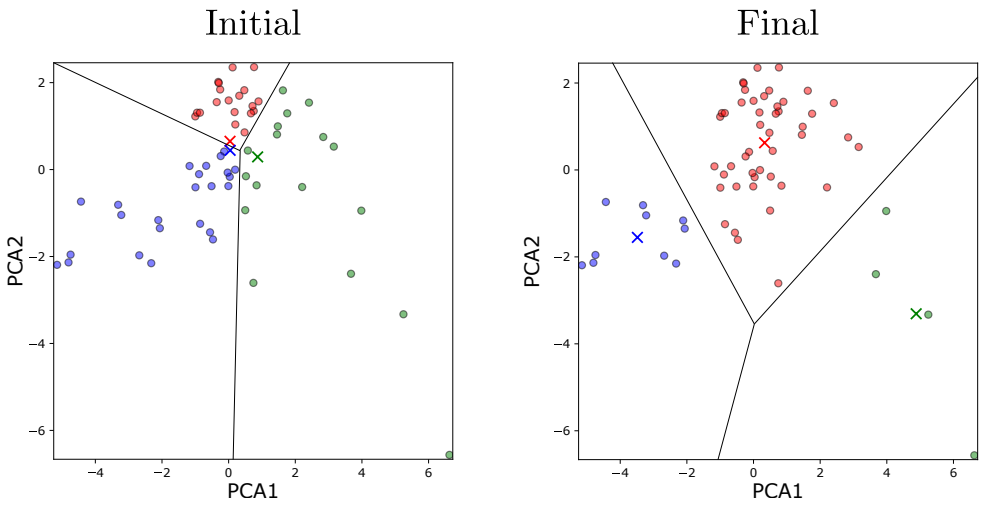

Figure 11.3: Evolution of the centroids for $K = 3$ and the same data as in the previous figures. The left and right panels show the same data on the PCA1 and PCA2 plane. The crosses indicate the initial (left) and final (right) positions of the centroids. The dots are coloured according to which cluster they correspond given the location of the centroids. The lines delimit the clusters. Between left and right panels the K-means algorithm was iterated 300 times. The improvement in the clustering from initial to final states is evident. See also Fig. 9.1.

The second comment relates to the is about the rigidity of the algorithm. When we calculate the sets, $\mathbf{S}$, we are using a strict rule for which cluster a data point belongs to. This rigid nature of the assignments of the data points to only one cluster each might work when there are clearly defined clusters. However, as with the city data example, when the clusters are less clearly defined one may question the effectiveness of these hard boundaries. We could relax this rigidity making the boundaries fuzzy; a method that does precisely this is called *fuzzy* K-means. Instead, let us try and compare these results to a different clustering algorithm without these rigid constraints.

11.2 Expectation Maximisation of Multi-Gaussian Models

We discussed above how a mixture of Gaussians can be used to model a data set's likelihood, see Sec.10.3. We now focus on a particular algorithm that tries to find the form of the Gaussian mixture

that maximises the expectation for data. This algorithm is called *Expectation Maximisation* (EM). The goal is to calculate the expectation of a data set $\mathbf{X}$ maximising the likelihood of that data.

Similarly to K-means clustering, we have to predetermine the number K of Gaussian that enter into the mixture, and the properties of each Gaussian, $\theta_k = (\vec{\mu}_k, \mathbf{\Sigma}_k)$. Here we have collected in the symbol θ_k the parameters that define each Gaussian, that is, its mean and covariance matrix[1], and we write the k-th Gaussian more compactly:

$$\mathcal{N}(\vec{x}|\vec{\mu}_k, \mathbf{\Sigma}_k) = \mathcal{N}_{\theta_k}(\vec{x}) \ . \tag{11.4}$$

The EM algorithm proceeds as follows:

1. Decide on the number of Gaussians K.
2. Initialise all the θ_k and the probabilities π_k.
3. For each data point $\vec{x}_i$ calculate the "responsibilities"

$$\gamma_{i,k} = \frac{\pi_k \mathcal{N}_{\theta_k}(\vec{x}_i)}{\sum_{l=1}^{K} \pi_l \mathcal{N}_{\theta_l}(\vec{x}_i)} \ , \tag{11.5}$$

   Loosely speaking the set of K responsibilities for a data point give an estimate to which cluster that data point belong to under the current parameters. Note the normalisation of the responsibilities,

$$\sum_{k=1}^{K} \gamma_{i,k} = 1 \quad \forall\, k \ . \tag{11.6}$$

4. Using the responsibilities compute the means

$$\vec{\mu}_k = \frac{\sum_{i=1}^{N} \gamma_{i,k} \vec{x}_i}{\sum_{i=1}^{N} \gamma_{i,k}} \ , \tag{11.7}$$

   and the covariance matrices

$$\mathbf{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_{i,k} (\vec{x}_i - \vec{\mu}_k) \cdot (\vec{x}_i - \vec{\mu}_k)^T}{\sum_{i=1}^{N} \gamma_{i,k}} \tag{11.8}$$

   This defines the new set of parameters θ_k for the Gaussians.

5. Calculate the weights

$$\pi_k = \frac{\sum_{i=1}^{N} \gamma_{i,k}}{N} \ . \tag{11.9}$$

   where the N in the denominator guarantees Eq.(10.2) via Eq.(11.6).

6. Repeat from 3 until convergence.

The EM algorithm tries to work out the best structure for the Gaussian mixture in order to increase the likelihood of the given set of data. It does this by giving each data point $\vec{x}_i$ an expectation $\gamma_{i,k}$ (i.e., the responsibility) of coming from the k-the Gaussian. This is reminiscent to K-means where each point is placed in a cluster, the difference being that in EM each point has a weight of belongs to each cluster given by the responsibility.

Furthermore, the steps where the parameters θ_k are calculated is reminiscent of the step in the K-means algorithm when the centroids are shifted. However, EM is doing more than just shifting the centre of the Gaussians. By recalculating the covariance matrices the shapes of the Gaussians are also changed to fit the data better. The same occurs with the updates of the weights π_k which alter the contribution of the different Gaussians to the mixture. A way to interpret the contribution of the k-th Gaussian is to think of it as representing the k-feature, with π_k giving the overall fraction of how much that feature contributes to the distribution.

[1] In ML parameters defining a model are very often denoted collectively by the symbol θ.

11.2.1 How well does it perform?

Given that we worked out the "best" number of clusters for the K-means algorithm as $K = 3$, we can use the same number of clusters or Gaussians for the EM algorithm. Also, as in K-means, we should explore multiple initial conditions for the parameters θ_k and π_k in order to ensure we are getting rid of any bias from initialising the algorithm.

We can see the results of the EM algorithm on the city data in Fig. 11.4. The centres of each cluster have settled in a similar place to those from the K-means algorithm, but the data is not clustered in quite the same way.

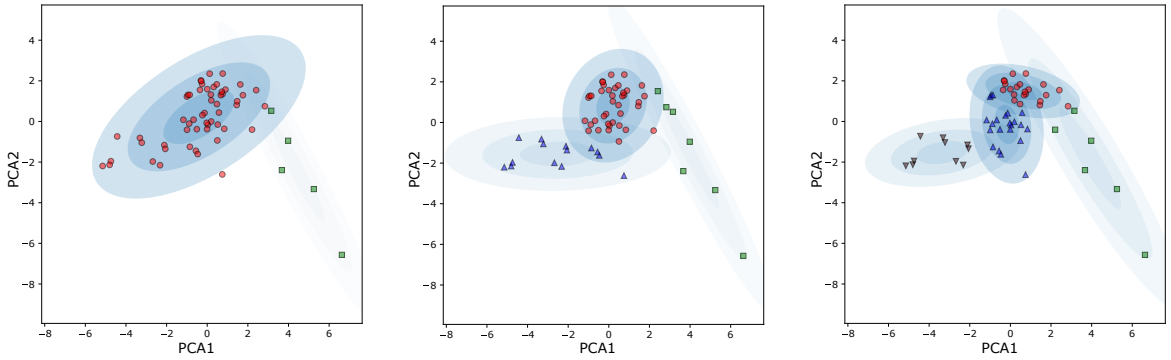

Figure 11.4: These are plots showing the result of trying to fit a mixture model of three independent Gaussian's to the city data. The difference between each plot is the number of Gaussian's in the mixture model, increasing from two (left) to four (right). Each cluster is shaped differently, and the data has a colour/shape based on which cluster most represents it. The contours show the Gaussian's for each cluster, with the first contour being one σ, the second two σ and so on. The deepness of the colour for each Gaussian represents its relative weight in the mixture, π_i.

There are a few key take away messages from this Fig. 11.4. Firstly, the data is coloured based on which cluster most represents it. We can see in the plot how the clusters are, for some data points, close in weight. Take for example the three triangle data points closest to the circles there is an influence from the both clusters here, but the one represented by the triangles is just stronger for these points.

Secondly, the weighting of each cluster is shown by the saturation of the colour. Clearly the cluster with the most data points in it, the red circles, has the least saturation. This makes sense, since this cluster corresponds to the largest increases in the likelihood of the data.

Finally, if we look at the effect of increasing the number of clusters we can see a substantive change between two and three clusters, but less of a change from three to four. This is in line with what we saw from the elbow analysis we did on the K-means clustering. Therefore, the most important information is most likely contained within three clusters. Four might be useful as well, but it is unlikely more than that will continue to be informative regarding new data. Especially if we remember that making the number of clusters too large can lead to over-fitting.

11.3 Similarities of EM and K-means

Given that the output of both of these clustering algorithms look similar, it is worth asking if they are related. The simple answer is yes. The EM of a mixture of Gaussians is a general form of which the K-means clustering is a specific case.

Let us put some constraints on the EM algorithm and see what happens. Firstly set the expectation $\gamma_{i,k}$ to take on a value of either one or zero, such that only one Gaussian k has value $\gamma_{i,k} = 1$ and the rest zero. This still satisfies Eq.(11.6). This is already very similar to the K-means case where the data is sorted out into sets using hard boundary rule, Eq.(11.1).

Next consider constraining the Gaussian to be "circular" by making the covariance matrices invariant under rotation in the plane. We discussed before how the K-means clustering worked by using the centroids as the centre of a circular bubble, see Fig. 10.1. If in the case of the GMM we choose the

covariant matrices to be proportional to the identity (and therefore rotationally symmetric)

$$\Sigma_k = r_k \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} , \tag{11.10}$$

where r_k can be thought of as the radius of the circle associated with the k-cluster, cf. K-means. If we write the corresponding GMM, it reads,

$$p(\vec{x}) = \sum_{k=1}^{K} \pi_k \frac{1}{(2\pi r_k^2)^{d/2}} e^{-\frac{|\vec{x}-\vec{\mu}_k|^2}{2r_k^2}} , \tag{11.11}$$

with $\pi_k = 1$ for the cluster with the most expectation on $\vec{x}$ and $\pi_{k'} = 0$ otherwise. That is, for each data point $\vec{x}$ there is only one term in the GMM. We denote the value of this k by $k(\vec{x})$. Equation (11.11) then reduces to

$$p(\vec{x}) = \frac{1}{\left(2\pi r_{k(\vec{x})}^2\right)^{d/2}} \exp\left(-\frac{|\vec{\vec{x}} - \vec{\mu}_{k(\vec{x})}|^2}{2r_{k(\vec{x})}^2}\right) . \tag{11.12}$$

The goal of GMM is to maximise the likelihood $p(\mathbf{x})$ of the data $\mathbf{x} = (\vec{x}_1, \ldots, \vec{x}_N)$. A crucial assumption is that the all the data points are independent (i.e., each data point is an independent sample of the problem under consideration). This means that the likelihood of $\mathbf{x}$ is the product of the likelihood of each data point

$$p(\mathbf{x}) = p(\vec{x}_1)p(\vec{x}_2)\cdots p(\vec{x}_N) , \tag{11.13}$$

where we have used that the joint probability of a set of independent variables is the product of the probability or each variable, and $p(\vec{x}_i)$ is given by Eq.(11.12) with $\vec{x}_i$ as its argument.

Maximising the likelihood Eq.(11.13) is equivalent to maximising the log likelihood,

$$\log p(\mathbf{x}) = \sum_{i=1}^{N} \log p(\vec{x}_i) = \sum_{i=1}^{N} \left(-\frac{d}{2}\log\left(2\pi r_{k(\vec{x}_i)}^2\right) - \frac{|\vec{x}_i - \vec{\mu}_{k(\vec{x}_i)}|^2}{2r_{k(\vec{x}_i)}^2}\right) , \tag{11.14}$$

with respect to the parameters θ_k, which are now the mean and the "radius" of each cluster, $\theta_k = (\vec{\mu}_k, r_k)$. Compactly, the optimal parameters θ^* are obtained from

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \left(\frac{d}{2}\log\left(2\pi r_{k(\vec{x}_i)}^2\right) + \frac{|\vec{x}_i - \vec{\mu}_{k(\vec{x}_i)}|^2}{2r_{k(\vec{x}_i)}^2}\right) , \tag{11.15}$$

where we have changed the maximisation to a minimisation by changing the sign. Notice that each data point belongs to one Gaussian so that the above expression can be rewritten

$$\theta^* = \arg\min_{\theta} \sum_{k=1}^{k} \sum_{\vec{x}\in\mathbf{x}:k(\vec{x})=k} \left(\frac{d}{2}\log\left(2\pi r_k^2\right) + \frac{|\vec{x} - \vec{\mu}_k|^2}{2r_k^2}\right) . \tag{11.16}$$

If we call n_k the number of points that belong to Gaussian k, then we can write the minimisation as

$$\theta^* = \arg\min_{\theta} \sum_{k=1}^{k} n_k \left(\frac{d}{2}\log\left(2\pi r_k^2\right) + \frac{\mathrm{var}_k(x)}{2r_k}\right) . \tag{11.17}$$

We see that this minimisation is very similar to that of K-means, Eq.(11.3).

Part III

Reinforcement Learning

Chapter 12

Introduction to Reinforcement Learning

Many people studying Machine Learning (ML) are also interested in the subfield of Artificial Intelligence (AI). For these people, the goal of AI is to someday produce Artificial General Intelligence, an entity which can learn or understand any intellectual task that a human being can. Applications of AI today generally have a very narrow focus, and excel at only one or, at most, a few tasks. While we are often reaching superhuman performance in an increasing amount of domains [9, 2, 8], these AIs cannot generalise cross-domain.

Supervised Learning (SL) at its heart, can only hope to solve tasks that are already able to be solved by humans in some form, whether that be personally (as in the case with image classification), or via expensive simulations or algorithms. SL relies on being able to give a learning agent information about correct or incorrect actions. One cannot use SL techniques to train a model to perform a task if no one knows how to label any data for that task. Unsupervised Learning (UL), similarly, can only uncover underlying trends in data. UL does not aim to have a model perform a certain behaviour, but simply, learn underlying patterns in supplied data.

This is where Reinforcement Learning (RL) comes in. This is a distinct branch of ML, very different to both SL and UL. RL aims to provide a framework for learning optimal, complex behaviour in scenarios in which we, as the teacher, are not able to provide optimal examples. RL is often deployed in scenarios where a model has to perform actions over multiple time steps, in order to achieve a certain goal. One can avoid giving examples of correct behaviour by instead constructing a scalar signal (known as the reward) which can inform the agent whether the actions taken were *good* or *bad*. This process of translating a complex task into reward signals can often vastly simplify the task. For example, knowing what is a good move to make in Chess is extremely difficult, but it is obviously bad if one loses and good if one wins. If one assigns a number to both winning and losing, say $+1$ and -1 respectively, then one can translate the task of "playing Chess well" into the problem of choosing an action in each situation that will maximise the chances of receiving a $+1$ reward.

The process of rewarding *good* behaviour and punishing *bad* behaviour explains the *Reinforcement* part of Reinforcement Learning, having some origins in the field of Psychology. One tries to reinforce *good* actions and discourage *bad* actions. This technique will seem familiar, as it is likely how you will have learnt to perform many tasks; e.g. if you make a mistake on a question, you alter your behaviour and learn how to avoid that mistake in the future. This is also the technique used to train a dog to perform tricks, by using treats to reinforce the desired behaviour. One does not simply teach the dog by performing the trick oneself, instead, one encourages the dog into that behaviour by coercing them with treats. In RL, we attempt to perform the same procedure, under the **reward hypothesis**:

> Any goal can be formalised as the outcome of maximising a cumulative, scalar, reward signal.
> (from the book by Sutton and Barto, Ref. [4])

This form of learning is done through *interaction with our environment*. This is often a very active form of learning as one can choose what information to gather through one's actions. Most importantly, using

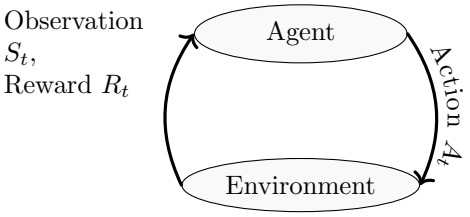

Figure 12.1: This diagram shows the typical setup of an RL problem. One usually depicts the environment and agent as separate, but it is important to remember that agents are often contained within the environment.

a reward signal for a goal definition can enable us to find optimal behaviour without any examples of optimal behaviour.

Reinforcement Learning is a term given to three distinct concepts:

1. A framework for mathematically defining a goal-oriented task as an RL problem.
2. The research field which studies the aforementioned problem.
3. The suite of algorithms that are developed to solve an RL problem.

Before we can dive into any more details, we must rigorously define what a reinforcement learning problem consists of. This is the topic of the next section.

12.1 Mathematical Definition

We have already outlined some of the ways one can think of defining a reinforcement learning problem. One usually refers to the diagram shown in Figure 12.1. A RL problem usually evolves in time, hence the cycle in the diagram. We usually denote the current time step as t.

The process starts with the agent observing the state of the environment. This can be denoted as $O_t = O(S_t)$, where $O(s)$ denotes the observation function which maps the state of the environment to the observation that the agent sees. For simplicity, we often assume that $O_t = S_t$, which is a *fully-observable environment*. When the agent receives the first observation, an internal policy function maps this observation to an action. In a deterministic case, this is usually written as $A_t = \pi(S_t)$, however, sometimes the agent follows a stochastic policy, which defines a probability distribution over all actions possible in the state s, given by $\pi(a|s)$ where we normalise the distribution:

$$\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1, \tag{12.1}$$

in which $\mathcal{A}(s)$ represents the set of actions that an agent can take in the state s.

Once the agent selects and carries out an action, it is applied to the environment. This, together with the previous state, is used to update the environment to the next time step. In a deterministic setting, one usually writes this as $s' = f(s, a)$, where s' is the next state.

Again, we can extend the framework to have a stochastic environment, whose function instead defines a probability distribution over all possible next states and rewards $p(s', r|s, a)$, where

$$\sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) = 1, \tag{12.2}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. Once a single transition has been made, a reward for the action taken can then be sampled using the reward function. In the deterministic reward setting, one can write this reward function as $r(s', s, a)$, which can be read as the scalar reward for transitioning from state s to state s' using action a. In the stochastic setting, the probability distribution is usually combined with that of the next state, which is shorthand expression given by

$$p(s', r|s, a) = f(s'|s, a)\phi(r|s', s, a), \tag{12.3}$$

where I have used $\phi(r|s', s, a)$ to define the probability distribution of receiving the reward r conditioned on transitioning from state s to s' using action a. This distribution is normalised as given by

$$\int_{-\infty}^{+\infty} \phi(r|s', s, a)dr = 1. \tag{12.4}$$

This reward function defines the goal the problem, and is usually constructed to influence the learning outcome (in a process known as *reward engineering*). However, this reward function can also be learnt via *inverse reinforcement learning* [7] based off of observed optimal behaviour. Like with the other parts of the process, this reward can also be stochastic in nature. The framework allows each individual process to have some stochasticity, however, pedagogically it is easiest to look at the deterministic case first.

We can now define a quantity, known as the **return**, which is the cumulative sum of rewards into the future, denoted as

$$G_t = R_t + R_{t+1} + R_{t+2} + \ldots + R_{T-1}, \tag{12.5}$$

where T indicates the time of reaching the **terminal** state. A terminal state defines an end to a trajectory, for which $R_{t'} = 0$ when $t' \geq T$. If a problem has a terminal state, we call it an **episodic problem**. One can alternatively specify non-episodic problems, which do not have any terminal state, but continue indefinitely. However, looking at eq. (12.5), one can see that this will become an infinite sum. One way to adapt this return, is to introduce a parameter γ, which controls how much an agent should pay attention to short term gains compared to long term gains. We introduce this parameter to the return as

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i R_{t+i}, \tag{12.6}$$

which is called the **discounted return**, as the parameter γ discounts future rewards in favour of short term rewards. This parameter is subject to $0 \leq \gamma \leq 1$. One recovers the episodic case by setting $\gamma = 1$ and setting all rewards from (and including) time step T equal to 0. We will use the term *return* to refer in general to the discounted return. Sometimes γ is omitted in the case that $\gamma = 1$.

One must remember that eq. (12.5) and eq. (12.6) are defined in terms of the actual sampled returns. They are just numbers, which are sampled from having an agent interact with the environment. We use the notation of a capital letter X to denote a sample of the quantity usually denoted by x. For example, you will see R_t for the reward sampled at time t and S_t as state sampled at time t, whereas r and s represent a generic reward and state respectively.

A more useful construct is called the **value function**, which calculates the expected return an agent will receive, given they are in the state s. This is usually written as:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega \sim \pi | S_0 = s}[G(\omega)], \tag{12.7}$$

where $\mathop{\mathbb{E}}_{\omega \sim \pi | S_0 = s}[A]$ denotes the average (or expected) value of the random variable A, which is sampled according to using the policy π, given that the initial state started at s. We use ω to denote a trajectory, which is made up of an array of state, action, reward tuples:

$$\omega = [S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_{T-1}, A_{T-1}, R_{T-1}, S_T], \tag{12.8}$$

where S_T is the terminal state. We use the notation $T(\omega)$ to indicate the terminal time step of the trajectory ω. In eq. (12.7), we are specifically saying we should run a trajectory starting at state s, and taking actions according to our policy π and transitioning states according to the environment and summing up the discounted reward G of the entire trajectory. The reason this is an expectation, is because if our environment is random (stochastic) in any way, even if our policy is deterministic, we can get different trajectories. The expectation is a way of averaging a random variable according to how likely we are to observe this random variable. Another way of writing an expectation is as a sum over all possible trajectories:

$$v_\pi(s) = \sum_{\omega} p_\pi(\omega) \delta_{S_0, s} G(\omega) = \sum_{\omega} p_\pi(\omega) \delta_{S_0, s} \sum_{t=0}^{T(\omega)-1} \gamma^t R_t, \tag{12.9}$$

where $p_\pi(\omega)$ represents the probability of that trajectories according to the policy π and the environment, and finally, only select those where the initial state S_0 is equal to s by using a Kronecker delta function (also called an indicator function) $\delta_{a,b}$ which equals 1 only when $a = b$ and 0 otherwise[1].

It is crucial to remember that value functions can only be defined in terms of a policy. Additionally, one implicitly makes the value function depend on the reward function and the environment function as well. Given that $G_t = R_t + \gamma G_{t+1}$, we can write eq. (12.7) as a recursive equation:

$$\begin{aligned} v_\pi(s) &= \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[r_t + \gamma G_{t+1}] \\ &= \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[r_t] + \gamma \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[G_{t+1}] \\ &= \sum_{a\in\mathcal{A}(s)}\sum_{s',r} p(s',r|s,a)\pi(a|s)r(s',s,a) + \gamma \sum_{a\in\mathcal{A}(s)}\sum_{s',r} p(s',r|s,a)\pi(a|s)\left[v_\pi(s')\right] \\ &= \sum_{a\in\mathcal{A}(s)}\sum_{s',r} p(s',r|s,a)\pi(a|s)\left[r(s',s,a) + \gamma v_\pi(s')\right], \end{aligned} \tag{12.10}$$

which states that the value of a state s is defined only in terms of the values of the states that s can transition to, and the reward received for moving between them.

In the deterministic case, this eq. (12.10) greatly simplifies to:

$$v_\pi(s) = r(f(s,\pi(s)),s,\pi(s)) + \gamma v_\pi(f(s,\pi(s))), \tag{12.11}$$

where the selected action in state s is given by $a = \pi(s)$ and the next state $s' = f(s,\pi(s))$, where f is the *environment function* which maps a current state and a given action to the next state. The reward is also awarded deterministically depending on the current state, the action taken and the next state.

One can also extend this idea to picking an arbitrary action, a, in the current state, but then following the same policy, π, for the rest of the trajectory. This allows us to compare immediate actions with one another. We can capture this idea in what is known as the **state-action value function**, which is given the symbol $q_\pi(s,a)$ and is defined as:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left[r(s',s,a) + \gamma v(s')\right], \tag{12.12}$$

and in the deterministic case as

$$q_\pi(s,a) = r(f(s,a),s,a) + v_\pi(f(s,a)). \tag{12.13}$$

This allows us to make the assertion that if $q_\pi(s,a) \geq v_\pi(s)$ then we can improve the policy π by taking the action a in the state s instead of the action $\pi(s)$. One can see that these definitions lend themselves to rigorously defining the quality of different policies, and reduce the problem to looking at single actions in single states, rather than considering entire trajectories.

The equations for $q_\pi(s,a)$ and $v_\pi(s)$ define what are known as **Bellman equations**, which form a large part of optimisation, known as **Dynamic Programming**.

To end this section, we will define the goal of Reinforcement Learning in these recently defined terms. We can state that the goal of RL is to find a policy π^*, which satisfies the following equation:

$$\pi^* = \arg\max_\pi v_\pi(s) \text{ for all } s. \tag{12.14}$$

One can also extend this definition to a set of initial states, as long as the probability distribution of being in that state is stationary and well-defined:

$$\pi^* = \arg\max_\pi \left[\sum_{s\in\mathcal{S}} d(s) v_\pi(s)\right], \tag{12.15}$$

[1]This assumes that the states take discrete values. For the case of continuous variables, the corresponding indicator function is called a Dirac delta function and written as $\delta(a-b)$.

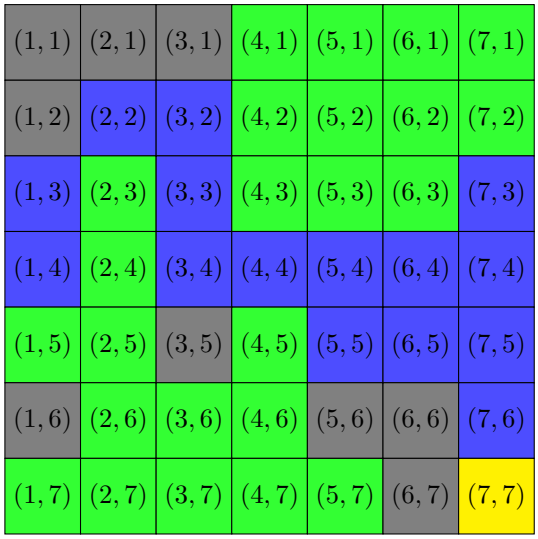

Figure 12.2: A representation of a grid world. Green represents grass, blue represents water, grey represents mountains and the yellow/gold colour represents the goal (or exit) tile. The labels of each tile shows the (x, y) position of the tile.

where $d(s)$ defines the probability distribution of state s being the initial state at $t = 0$. This definition reduces to eq. (12.14) in the case that $d(s)$ is uniform across all states. Stated in another way, the optimal policy π^* is the policy, or set of policies which obey

$$v_{\pi^*}(s) \geq v_\pi(s) \text{ for all } s \in \mathcal{S} \text{ and } \pi. \tag{12.16}$$

In words, eq. (12.15) states that the goal of RL is to choose an optimal policy, which maximises the expected cumulative (total) reward of an agent when interacting with an environment.

12.2 Grid World

When we are talking about reinforcement learning, it is helpful to have a go to example to which we can apply the theory. A very simple environment/problem that is easy to think about is the task of learning to navigate through a 2-dimensional grid world, as shown in Figure 12.2, in order to reach a target. We can express the goal of trying to reach the target in the shortest amount of time. Each different tile in the grid world takes a different amount of time to cross. We can think of each different tile as a different type of terrain, like on a map. Green represents grass, the easiest tile to cross, taking 1 unit of time to cross. Blue represents water, which takes 3 units of time to cross. There are the grey tiles which represent mountains, and take 5 units of time to cross. Finally, there is the gold tile, which represents the target destination in the grid world. In this simplified world, an agent can only move to adjacent tiless, and the "game" ends when the agent reaches the gold tile.

Using what we have learnt in the previous section, we can translate this problem into a reinforcement learning problem. We can start with the state of the problem. In this case, the state would be the position of the agent in the grid world. One can label the east-west (horizontal) and north-south (vertical) directions as x and y respectively. We will assume for now that the terrain map does not change, which means we do not have to include it in part of the state[2], as it becomes an implicit part of the problem. We can denote the state as $s = (x, y)$.

The set of actions of an agent can be denoted by $\mathcal{A} \in \{\uparrow, \downarrow, \rightarrow, \leftarrow\}$ for north, south, east and west respectively. The action which maps from state s to the next state s' is denoted as a. The environment

[2]If we were training an agent to operate in any grid world configuration, we would have to include the details of the tile configuration in the state.

(a) (b)

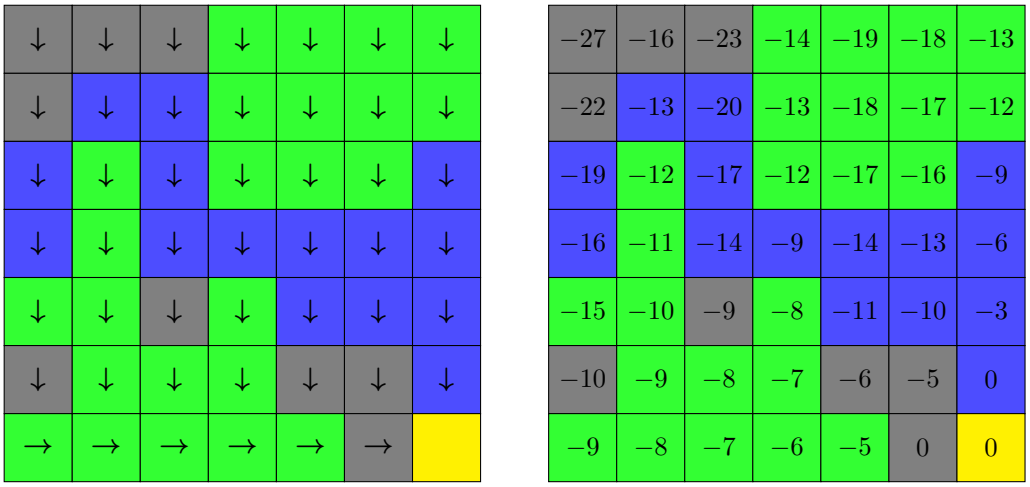

Figure 12.3: (a) shows the initial policy, π_0, in grid world. We do not allow actions that would cause the agent to exit the grid world shown. (b) shows the initial value function, v_{π_0}, in grid world. This value function is only defined in terms of the policy π_0, shown in (a).

function is deterministic and determines how s is mapped to s' using a:

$$s' = f(s,a) = \begin{cases} (x, y-1), & a \text{ is } \uparrow \\ (x, y+1), & a \text{ is } \downarrow \\ (x+1, y), & a \text{ is } \rightarrow \\ (x-1, y), & a \text{ is } \leftarrow \end{cases}. \tag{12.17}$$

Our final choice is the reward function. How are we to encode the goal of reaching the gold tile in the shortest possible time? The goal of a RL problem is to maximise a reward. As our grid world goal is to minimise the time, we can instead maximise the "negative" of the total time taken, which is an equivalent goal. We can encode the cost to travel on a given tile as:

$$c(s) = \begin{cases} 0, & s \text{ is a goal (gold)} \\ 1, & s \text{ is grass (green)} \\ 3, & s \text{ is water (blue)} \\ 5, & s \text{ is mountains (grey)} \end{cases}, \tag{12.18}$$

where s represents the tile, indicated by an (x, y) position on the grid. We can encode the travel time as a negative reward for travelling into that tile:

$$r(s', s, a) = -c(s'), \tag{12.19}$$

where the reward is deterministic and only depends on the tile that is travelled into. In general, the reward function can depend on any number of variables, but in this case, it only depends on the next tile entered. As an example, if a player moves from a grass tile (green) into a mountain tile (grey), they will receive a reward of -5. In fact, moving from any tile into a mountain tile will receive a reward of -5.

We can represent a deterministic policy with arrows in each tile, indicating the direction of travel from that tile, as shown in Figure 12.3(a). If we imagine starting at $(4, 6)$, we can see that the path followed is:

$$\omega_{\pi_0}(S_0 = (4,6)) = \{S_0 = (4,6), A_0 = \downarrow, R_0 = -1 \tag{12.20}$$
$$S_1 = (4,7), A_1 = \rightarrow, R_1 = -1 \tag{12.21}$$
$$S_2 = (5,7), A_2 = \rightarrow, R_2 = -5 \tag{12.22}$$
$$S_3 = (6,7), A_3 = \rightarrow, R_3 = 0 \tag{12.23}$$
$$S_4 = (7,7)\}, \tag{12.24}$$

where we set the starting time at $t = 0$, the starting state at $S_0 = (4, 6)$ and follow the policy π_0 until the terminal goal state, giving us a trajectory, $\omega_{\pi_0}(S_0 = (4, 6))$. We can calculate the return of the trajectory as $G_0 = R_0 + R_1 + R_2 + R_3 = -7$, using $\gamma = 1$. As we are in a deterministic setting, we know that $v_{\pi_0}(S_0 = (4, 6)) = G_0 = -7$ since there is only one possible trajectory starting at $(4, 6)$.

We can repeat this type of calculation, starting at each tile on the map, to calculate the value of each state under the initial policy. However, this method is very inefficient, as many trajectories contain sub-trajectories which may have already been evaluated. Instead, we can use the recursive form of the value function to calculate each value by only visiting each state a single time.

We start by labelling the terminal state (gold) as having a value of 0, by definition, as the episode ends here and there are no more subsequent rewards. If you look carefully at the policy, one can see that there are no cycles and one always ends in the terminal state, regardless of starting position. For this reason, we can use this initial value, along with eq. (12.11), to "propagate" the values backwards so that each state has a corresponding value (under the policy π_0). We then visit all the tiles which are adjacent to the labelled states, updating those that have actions that take an agent into one of our already labelled state, using eq. (12.11).

Under this new method, after giving the initial value to the terminal state, we visit states $(6, 7)$ and $(7, 6)$, which also have a value of 0, since you get a 0 reward for entering the terminal state, and the value of the terminal state is also 0. Next of all, we visit $(5, 7)$, $(6, 6)$ and $(7, 5)$. They have actions taking an agent into a currently labelled state; the values of these next states are -5, -5 and -3 respectively, as the first two enter a mountain tile and the last enters a water tile. This process is iterated, until all tiles are labelled. The final result of this process is shown in Figure 12.3(b).

Notice how both methods we chose would give identical results, except the second is much more efficient at evaluating every state. The efficiency comes from breaking the problem down into a single state transition, and choosing a sensible order in which to evaluate the states. The order in which we calculate the values is called a **sweep**.

12.3 Bellman Equations and Optimality

In Section 12.1, we defined equations 12.11 and 12.13 for v_π and q_π respectively. We called these the **Bellman Equations**. These two functions are very important to RL as they provide a **partial ordering** of the quality of policies. A policy, π, is defined to be "better" than or equal to a policy, π', if the expected return is greater than or equal to than that of π' for all states [10]. Put in an equation:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}, \tag{12.25}$$

where $\mathcal{S}$ defines the state space of the problem, and $\iff$ is the symbol for *if and only if*. This is only a partial ordering, since it does not disambiguate two policies with the same values in each state, even if the policies are different. However, this condition is enough to ensure that we can define at least one optimal policy, denoted as π^*. Even though there may be multiple optimal policies, they all share the same state-value function, called the **optimal state-value function**, denoted as v^*, and defined as

$$v^*(s) \equiv \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}. \tag{12.26}$$

Optimal policies also share the same **optimal state-action value function**, denoted as q^* and given by

$$q^*(s, a) \equiv \max_\pi q_\pi(s, a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s). \tag{12.27}$$

One can also write the above equation in terms of v^*:

$$q^*(s, a) \equiv \sum_{s', r} p(s', r|s, a) \left[r + \gamma v^*(s')\right]. \tag{12.28}$$

One can see that this is a strictly stronger set of conditions for defining optimality than eq. (12.15), as it does not define an initial state (or distribution of initial states). The difference between these equations is that if we restrict our search space of policies to one which does not explore part of the possible state space (as it is not optimal), then a policy which is not optimal in this area is still optimal in terms of the

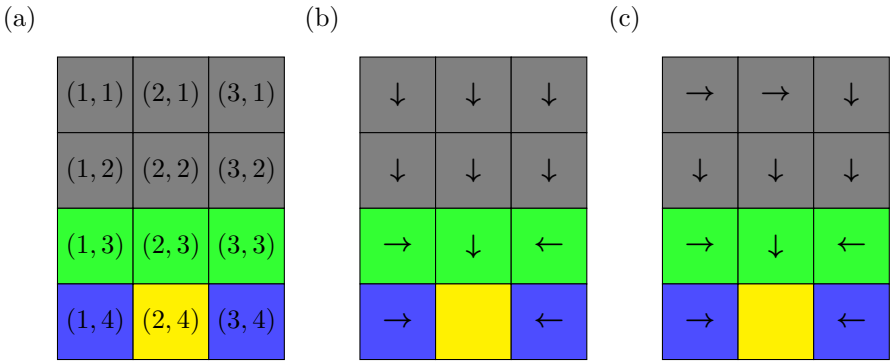

Figure 12.4: (a) shows the map, along with the coordinates in the form of (x, y). (b) shows the optimal policy w.r.t eq. (12.26). (c) shows a policy which is not optimal in terms of eq. (12.26) but is optimal in terms of eq. (12.15), in the case that $d(s) = 0$ for $y = 1$. This means that the probability of starting at the top of the map is zero, and hence, the optimal policy will never visit any of those states, so it is still an optimal policy when we take into account the initial states.

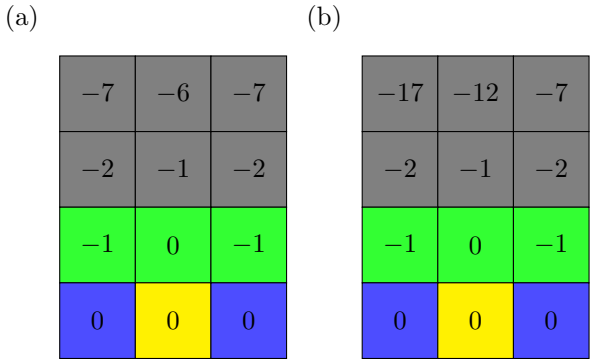

Figure 12.5: (a) and (b) show the value functions for the policy in Figure 12.4 (b) and (c) respectively. The values for the visited states under an initial starting state distribution of $y > 1$ are identical for both policies.

goal we care about (the initial position). We can use the grid world example from before to understand this.

Picture the grid world shown in Figure 12.4(a). If we were to follow the definition of optimality given by eq. (12.26), then we would arrive at an optimal policy given by Figure 12.4(b). However, if we know that we only start in tiles where $y > 1$ (i.e. excluding the top line of the map), we know that it is optimal to avoid this line. As the agent never enters the state space in the top row, the policy here is irrelevant, as optimality can be defined in terms of the starting state, given by eq. (12.15). We can see that for all states actually visited by the policies, the values are identical. However, we do not always know which states will not be visited by an optimal policy starting at a distribution of initial states, and as such, we should prefer the more general definition of optimality. This ensures that we do not discard part of the state space which may be essential to finding the optimal policy.

12.4 Policy Evaluation

In our grid world example, we presented a few ways of calculating the values for a given state. These methods relied on the fact that we knew where the exit tile was, and knew that it would be a terminal state. This sort of information is problem specific, and the dynamics of the environment are not always known, but need to be explored. It is therefore important that we discuss a more general way of constructing the values for a given policy.

12.4.1 Trajectory Evaluation

The first method that we should discuss is directly evaluating the returns of a trajectory from a state. By definition in eq. (12.7), we know that the value of a state is the expected discounted return of a trajectory starting at that state. If we wish to estimate this value, we can just run trajectories starting at the state s, and take an average. In the episodic and deterministic settings, such as grid world, we can just run a single trajectory starting at s until termination and calculate the return by adding up the discounted rewards. In a non-deterministic environment, or with a stochasitc policy, one has to run many trajectories and take an average. This average can be estimated to any arbitrary precision at the cost of more trajectories. In the stochastic case, running trajectories to estimate quantities like returns is a **Monte-Carlo** process, common throughout the RL literature.

This approach is clearly very computationally expensive and does not benefit from the recursive nature of the Bellman equations and requires re-calculating values for a large state space. Additionally, in the non-episodic case, where $T \to \infty$, one usually has to settle for an estimate for the state, taking only $T_{\max}$ steps in the trajectory, where $T_{\max}$ is finite. Remember that $\gamma < 1$ and so the t^{th} step in the trajectory has a prefactor of γ^{t-1}. If the rewards are static and finite, then the contributions from additional terms tend towards 0, making the approximation arbitrarily accurate.

12.4.2 Exact Value Calculation

If the dynamics of a problem are entirely known, then we have a system of equations, one for each state s, given by

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]. \tag{12.29}$$

We have $|\mathcal{S}|$ unknowns (i.e. the number of states) and all equations are linear in the unknowns. This can be solved as a simultaneous equations problem. The dynamics is entirely given by the probability distribution $p(s', r|s, a)$ and the policy π.

12.4.3 Value Iteration

Instead of performing many trajectories, one can instead use the Bellman equations to iterate the state values until they have converged. To start, we initialise a set of values for each state. This does not need to be a correct value, so a good choice is to usually set the values of each state to zero, as all terminal states will have the correct value, despite not knowing which states are terminal ahead of time. As this will be an iterative process, we will use the superscript (k) to specfiy the k^{th} iteration of a quantity. Our initial conditions are:

$$V_\pi^{(0)}(s) = 0 \text{ for all } s, \tag{12.30}$$

where we use the symbol V_π to denote the approximate value function, whereas v_π is reserved for the true value function.

The only condition on our initialisation is to make sure that terminal states are set to 0. As we are initialising all states at 0, this ensures the terminal states are also initialised at 0. The way we update the value functions is by applying the Bellman equations:

$$V_\pi^{(k+1)}(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi^{(k)}(s')\right] \text{ for all } s, \tag{12.31}$$

which, in the deterministic case, becomes:

$$V_\pi^{(k+1)}(s) = r\left(f[s, \pi(s)], s, \pi(s)\right) + \gamma V_\pi^{(k)}\left(f[s, \pi(s)]\right) \text{ for all } s, \tag{12.32}$$

where $f[s, \pi(s)]$ is the state transitioned to, from s, using the action selected from the policy. One should pay attention to the fact that we update all states in one go, and only use values from the previous iteration to produce the next iteration.

This process will iterate towards the true value function. In the case of continuing problems, then one iterates until differences in subsequent iterations are within a chosen, arbitrary, threshold. We can apply this on the grid world problem in Figure 12.4(a), using the policy in Figure 12.4(b). Figure 12.6 shows the iterations of this happening until there are no more differences, starting from $k = 0$, until $k = 5$.

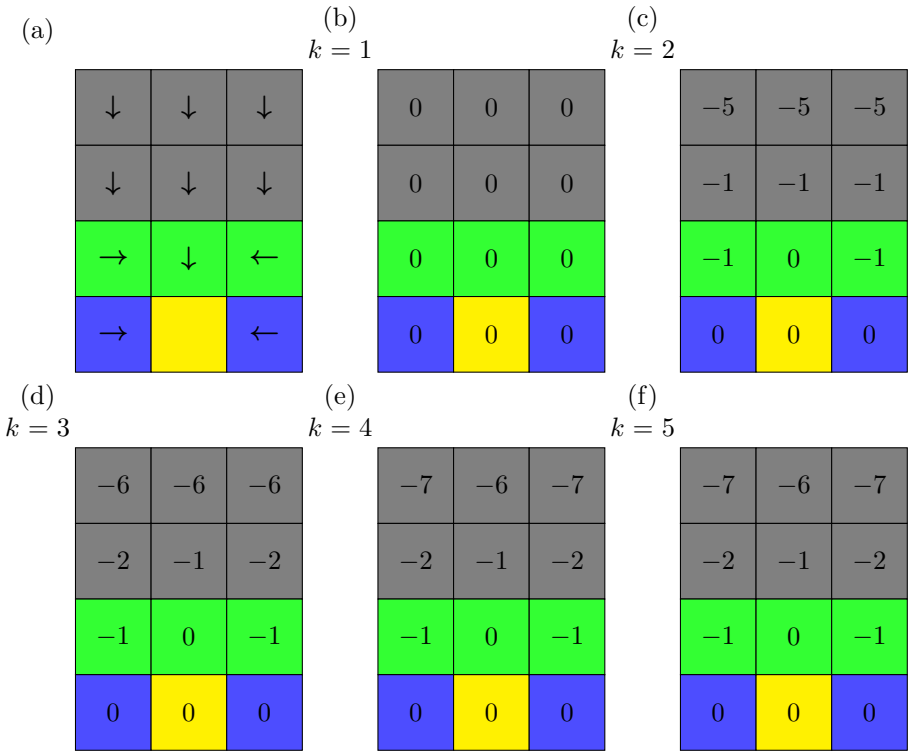

Figure 12.6: (a) shows a given policy π, same as Figure 12.4(a). (b)-(f) shows the value iteration sweeping over all states using eq. (12.31). A final round of value iteration is shown in (f) to confirm that this is a fixed point and does not change.

Notice that $k = 4$ and $k = 5$ are identical, so the iteration process ends. One does not know the process has finished until there are no changes. We can see that the final value function here is the same as the one presented in Figure 12.5(a).

The process of value iteration is guaranteed to converge to the correct answer, provided that each state is visited infinitely often. This is because eq. (12.31) clearly has a fixed point when $V_\pi^{(k)} = V_\pi$.

12.5 Policy Iteration

Now that we can calculate the value function, we can equally calculate the *state-action value function*, via:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r(s', s, a) + \gamma v_\pi(s') \right], \tag{12.33}$$

which in the deterministic regime becomes eq. (12.13).

Using $q_\pi(s, a)$ allows us to compare a single transition action a in the state s. In fact, if we have a policy π', we can prove that it is better than a current policy, π, under the condition that

$$v_{\pi'}(s) \geq v_\pi(s) \text{ for all } s. \tag{12.34}$$

One can see that if we create a new policy such that

$$\pi'(s) = \max_a q_\pi(s, a) \text{ for all } s, \tag{12.35}$$

then we are guaranteed to satisfy eq. (12.34). This will be expanded on in the problem sheets.

We can use eq. (12.35) to generate a new, improved, policy, π'. In turn, we can use any method to evaluate the new value function, $v_{\pi'}$, and then continue the process. We can see that the optimal policy, π^*, is a fixed point of this process. For the same reason that the value function iteration is guaranteed

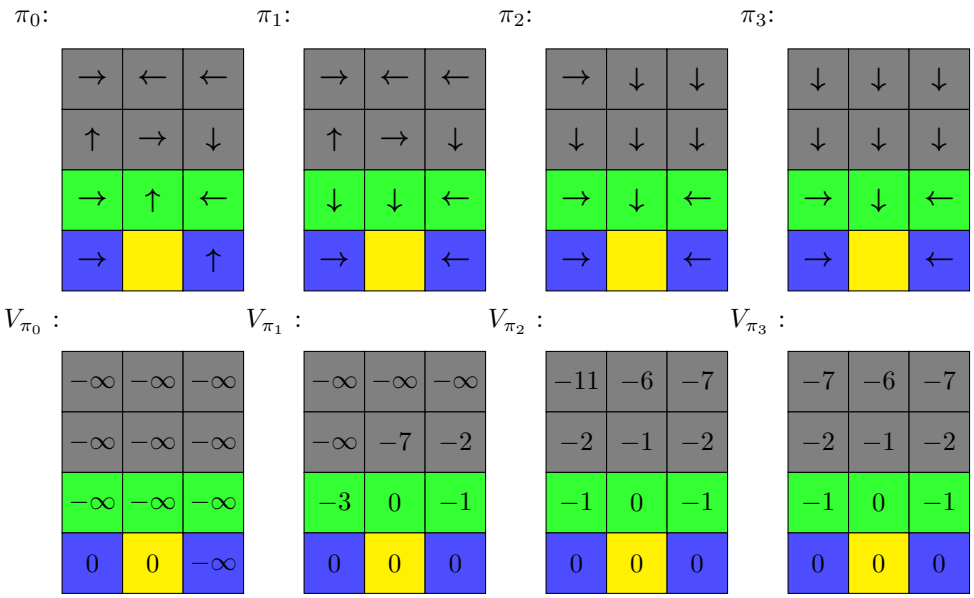

Figure 12.7: Shows the process of generating an optimal policy from a randomly initialised policy via policy iteration. Values are calculating by running a trajectory, and detecting loops (to calculate $-\infty$ values), instead of the value iteration method due to efficiency. The final optimal policy generated is the same as Figure 12.4(b). The final policy iteration is not shown, as it is the optimal policy.

to converge, this process is also guaranteed to converge to the optimal policy. This allows us to solve most simple problems exactly.

The policy iteration algorithm to find an optimal policy is summarised below:

1. Generate an initial policy π_0 and set $n = 0$, where π_n is the n^{th} policy.
2. Initialise $V_{\pi_n}^{(k)}(s) = 0 \; \forall \; s$ and set $k = 0$.
3. Calculate $V_{\pi_n}^{(k+1)}$ using eq. (12.31), sweeping over all states.
4. Check for convergence between $V_{\pi_n}^{(k+1)}$ and $V_{\pi_n}^{(k)}$. If converged, set $V_{\pi_n} = V_{\pi_n}^{(k+1)}$ and continue, else go to step (3).
5. Create a new policy, π_{n+1} using eq. (12.35) based off π_n.
6. Stop if $\pi_{n+1} = \pi_n$ and output $\pi^* = \pi_n$ and $V^*(s) = V_{\pi_n}$.
7. Increment n, such that $n \leftarrow n + 1$ and go to step (2).

We can substitute any valid policy evaluation technique to calculate v_π into this algorithm.

Let's see this algorithm in practice, by applying it to a random policy and performing iterations on this. We only show the final value function for each policy, as the process is already detailed in Figure 12.6.

12.6 Summary

In this chapter, we have covered the basic terminology of reinforcement learning, and formulated a simple grid world game as a RL problem. We have explored exact methods of solving these problems using the Bellman equations and dynamic programming. In the previous section, we gave a basic algorithm for finding the optimal policy of a generic reinforcement learning problem.

However, the field of RL rarely focuses on the methods presented in this chapter despite being the bedrock of the field. One of the reasons for this is due to a lack of a model for the environment, rendering dynamic programming techniques unsuitable. Additionally, many problems that RL is applied to have an incredibly large state and/or action space, which makes these methods computationally infeasible. In the next chapter, we will discuss ways of learning, without explicit access to a model, but being able to interact with the environment to generate experience, via *model-free methods*.

Chapter 13

Model-Free Methods

In the previous chapter, we assumed that one could solve problems by exhaustively searching the entire state space, and confidently knowing how the environment would respond to actions; we call this having a *model* of the environment. These assumptions are not always feasible. Imagine the scenario of training a self driving car. In this scenario, the behaviour of other objects in the environment, such as people, cannot be effectively contained in a model of the environment. There are some methods that try to learn a model of the environment to help with prediction, but it is common to have access to no model at all.

Additionally, most problems have a *combinatorial* state-space, e.g. backgammon has over 10^{20} states [10]. Say that one could evaluate a million states per second, it would still take over 1000 years to complete a single sweep. This is not even mentioning problems with continuous states or actions, which have an infinite state space. In practice, these are usually discretised or approximated, however, one can see how the state space could grow arbitrarily large.

It should be stated that the dynamic programming methods described previously, while impractical for large problems, are actually quite efficient. The time taken to reach an optimal policy, in the worst case, is polynomial in the number of states and actions. If a problem has n states and k actions, then it is clear that one can choose k actions in each state n, and as such, the number of total policies is k^n. This is an exponential, in state space, number of policies; however, dynamic programming will solve this in polynomial time. This is wildly efficient, nethertheless, if n is too large to sweep over, then it is infeasible. In the real world, many problems fit into this category.

One way in which we can focus the search in policy space is to learn through experience, focusing on the part of the state space that an agent is likely to visit and be important to finding the optimal policy. These are collectively known as **Monte-Carlo Methods** and form the basis of most modern reinforcement learning algorithms. Unlike the previous chapter, we do not assume any knowledge of the environment (i.e. the transition function/rates), making the techniques *model-free*. A typical Monte-Carlo method only requires *experience* - a sample sequence of states, actions and rewards from actual or simulated interaction with an environment.

13.1 Policy Evaluation through Interaction

In Section 12.4.1, we discussed using trajectories to evaluate the value of a given state. While this is not the most efficient approach in some problems, such as grid world, Monte-Carlo trajectories form a very important part of modern RL, as DP methods are often infeasible.

To remind ourselves, the definition of the value of a state is

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega\sim\pi|S_0=s}[G(\omega)], \tag{13.1}$$

where $G(\omega)$ is the return (sum of discounted rewards) of a trajectory beginning at state s. One can imagine running a trajectory beginning at some initial state s drawn from a distribution of initial states with probability $d(s)$. The quality of a policy is given exactly by the expected discounted return of

trajectories generated by the starting distribution and the policy π. We can write this down in terms of the value of the initial states:

$$\langle G_\pi \rangle = \sum_{s \in \mathcal{S}} d(s) V_\pi(s) = \sum_{s \in \mathcal{S}} d(s) v_\pi(s). \tag{13.2}$$

As before, this provides a *partial ordering* over all policies, such that if $\langle G_{\pi'} \rangle > \langle G_\pi \rangle$, then $\pi' > \pi$. It is only a partial ordering, since if $\langle G_{\pi'} \rangle = \langle G_\pi \rangle$, then both policies are equal and cannot be distinguished with this metric. Notice that this is a slightly different metric to the Bellman policy ordering (given in eq. (12.25)), as it compares the weight of states, and does not require iterating over the entire state space to compare policies. We have relaxed the restriction that the policy must have the most optimal value function over *all* states, but instead focus on a weighted average of the values of the initial states. A true optimal policy (over all states) is also an optimal policy with respect to an initial distribution, but vice versa is not neccessarily true.

In the dynamic programming approach of policy evaluation, one had to visit every single state of the problem iteratively. For many problems this can be prohibitively expensive as the state space of most problems is very large. Instead, one can consider which states are actually relevant to solving the problem, as some states may not even be reachable depending on the initialisation conditions. We touched on this in previous chapters, shown in Figure 12.4 and Figure 12.5, where under the initialisation conditions of $y > 1$, these two policies have the same expected returns, as the trajectories from the initial states never enter the regions where the policy differs. Often, optimisation of dynamic programming algorithms comes in the form of biasing the sweep of states to those which are more likely to be relevant to solving the problem. Take the example of finding the shortest path between two points. The default algorithm for this is called "Dijkstra's algorithm", which is often augmented for the A* algorithm, which prioritises searching in the direction of the target, as it deems these states *more relevant*.

Along with prioritising important states, interaction with the environment does not require the agent to have a model of the environment to learn. Dynamic programming explicitly requires a model of the environment to learn, which we called a *model-based* method. This usually also involves methods that have some aspect of *planning*, which allow an agent to plan out future actions, much like a Chess player visualising the board a few moves into the future[1].

In order to evaluate a given state, one can simply run a number of trajectories, $\omega_\pi^{(i)}$, using the current policy, π, starting at a state s to approximate the value of that state:

$$V_\pi(s) \approx \frac{1}{N} \sum_i^N G(\omega_\pi^{(i)}), \tag{13.3}$$

where $G(\omega)$ is the total return of the trajectory ω, and N is the total number of samples. eq. (13.3) becomes an equality in the limit of $N \to \infty$. Notice that one can sample ω without having to know the specific model of the environment. Instead, one need only be able to interact with the environment and does not need to know the explicit probabilities of certain transitions and rewards.

The algorithm for estimating the value function $V(s) \approx v_\pi(s)$ using Monte-Carlo is given in Algorithm 1. It should be noted that this is called the *every-visit* Monte-Carlo method, which averages returns from every single visit to the state s. An alternative approach is to only alter the average to $V(S_t)$ if it is the first visit in the trajectory, known as the *first-visit Monte Carlo method*, discussed in more detail in Chapter 5 of [10].

13.2 Monte-Carlo Policy Iteration

Estimating state-values tends to be less helpful when one does not have access to the model of the environment. Previously, we could construct a greedy policy, using a value function $V(s)$ using

$$\pi(s) = \arg\max_a \left[\sum_{s' \in \mathcal{S}} p(s', r|s, a) \left(r + \gamma V(s') \right) \right], \tag{13.4}$$

[1] Planning is not extensively covered here, but Chapter 8 of [10] provides a basic overview. Planning is also extensively used in *Monte-Carlo Tree Search* (MCTS), which is one of the main algorithms behind AlphaGo [9], the first computer program to beat a world champion at the game of *Go*.

Data: Policy π
Result: Estimated state-value function $V(s) \approx V_\pi(s)$.
Initialise a state-value-function $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$;
Initialise an empty list, $Returns(s)$ of sampled returns for all $s \in \mathcal{S}$;
for $i \leftarrow 1$ **to** N_{max} **do**
  Sample a trajectory, ω, using π: $\omega = (S_0, A_0, R_0 \ldots, S_T, A_{T-1}, R_{T-1})$;
  $G \leftarrow 0$;
  for t *in* $T-1, T-2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_t$;
    Append G to $Returns(S_t)$;
    $V(S_t) \leftarrow$ average$(Returns(s))$;
  end
end

Algorithm 1: Value-Function estimation using Monte-Carlo policy evaluation.

where $p(s', r|s, a)$ is the model of the environment. However, without a model, we cannot compute this. Instead, we should attempt to estimate the state-action value function, $q_\pi(s, a)$, for a given policy. One approach to estimating the state-action value function is to randomly start in any state and select a random starting action. After this first transition, one follows the given policy π until the end of the trajectory. One can then calculate the return for the starting state, and action and use this as an estimate for the q_π value at the initial state, using the initial action. This is known as *exploring starts*. We do not consider this approach further, as it is prohibitively computationally expensive in most cases.

Instead, we consider making our policy stochastic, to ensure there is some exploration. One way to make a policy stochastic, is to make it *ϵ-greedy*, which means that with probability $(1 - \epsilon)$, we pick the actions according to $a^*_\pi = \arg\max_{a'} q_\pi(s, a')$, and otherwise, choose a completely random action. The probability of this can be defined as

$$\pi(a|s) = \begin{cases} (1-\epsilon) + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = a^*_\pi \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq a^*_\pi \end{cases} \tag{13.5}$$

We choose ϵ to be between 0 and 1. Notice that we recover a greedy policy if we set ϵ to zero. Usually, one anneals ϵ towards zero to decrease exploration and maximise exploitation at the end of training.

Data: Soft policy π
Result: Estimated state-action-value function $Q(s, a) \approx q_\pi(s, a)$.
Initialise a state-action-value-function $Q(s, a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$;
Initialise an empty list, $Returns(s, a)$ of sampled returns for all $s \in \mathcal{S}$;
for $i \leftarrow 1$ **to** N_{max} **do**
  Sample a trajectory, ω, using π: $\omega = (S_0, A_0, R_0 \ldots, S_T, A_{T-1}, R_{T-1})$;
  $G \leftarrow 0$;
  for t *in* $T-1, T-2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_t$;
    Append G to $Returns(S_t, A_t)$;
    $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$;
  end
end

Algorithm 2: State-action value function of the soft policy π, which ensures that all actions are taken with some finite, non-zero, probability.

We can now evaluate the state-action value function of the exploring policy π, using Algorithm 2. Once the policy has been estimated, one can iterate the policy such that $\pi^{k+1}(a|s) = \arg\max_{a'} Q^{(k)}(s, a')$ for all $s \in \mathcal{S}$, similar to the policy iteration shown in Section 12.5. This iteration is guaranteed to converge to the optimal soft policy. One can also imagine this soft policy as always choosing the greedy action, but the environment randomly (with probability ϵ) overriding the agent's action with a uniformly random one instead. In this case, the best that the agent can do is choose an action which maximises the state-action-value function, which takes into account this action randomisation.

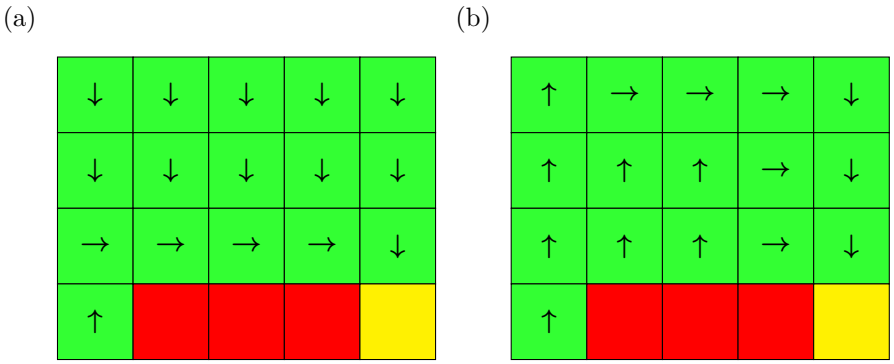

Figure 13.1: A grid world example with lava blocks at the bottom. A soft-policy has a small chance of entering the lava if the agent is in a tile near it. Imagine the environment has a strong wind that will move the agent at random into another tile with a small chance. In this situation, the only optimal policy is one which takes you far away from costly tiles, as these states will have a lower value as there is a contribution from the random chance that you are knocked into the adjacent costly tile, whatever your desired action. This example illustrates that the optimal hard policy, shown in (a), is not the optimal soft policy (b).

13.3 On/Off Policy Methods

Notice in the last section we presented learning a *soft*-policy[2] π for optimisation, ensuring that samples have some exploration. However, this policy can be quite suboptimal. We will use an example from grid-world to emphasise this.

Let's introduce a new type of tile to map and call it "lava". This will symbolise a type of tile that is to be strongly avoided, and let's give it a cost of some high value, say 100, meaning that the agent will receive a -100 penalty for entering the tile, causing the episode to end. If we have an *ϵ-greedy* policy, then there is a chance the agent will randomly enter the tile if it is adjacent. For this reason, there is a strong influence to stay far away from any lava tiles, even if the optimal path to the exit goes near them.

Take the policies given in Figure 13.1, given a starting point of the south-east corner, we can clearly see that the optimal path is to avoid the lava, as shown in (a). However, if we were to instead calculate the optimal *ϵ-greedy* (soft)policy, we can see that this will diverge from the "true" optimal path, and we pay the price for baking in exploration to the existing policy.

To keep exploration, but ensuring we learn the optimal policy, we can modify our algorithm to be *off-policy*. An *off-policy* algorithm uses a different policy to generate trajectories, from which we can learn the state action value function of our actual policy π, then used for policy iteration. In general, we can call this other policy, b, which is used to generate sample trajectories to learn from.

Imagine that we have a trajectory ω, generated using b, which has a probability $p_b(\omega)$. Under a different policy π, this probability would be $p_\pi(\omega)$. Remember that we can calculate the expected value of the return (under π) via

$$\langle G \rangle_\pi = \mathop{\mathbb{E}}_{\omega\sim\pi}[G(\omega)] = \sum_\omega p_\pi(\omega) G(\omega), \tag{13.6}$$

which assumes that ω starts in a state s with probability $d(s)$.

[2] A soft policy is one which is stochastic, and takes every available action a with at least $\pi(a|s) > 0$.

We can multiply each term inside the sum by $\frac{p_b(\omega)}{p_b(\omega)}$, as this is equal to 1:

$$\begin{aligned}\langle G\rangle_\pi &= \sum_\omega p_\pi(\omega)G(\omega)\\ &= \sum_\omega \frac{p_b(\omega)}{p_b(\omega)}p_\pi(\omega)G(\omega)\\ &= \sum_\omega p_b(\omega)\left[\frac{p_\pi(\omega)}{p_b(\omega)}G(\omega)\right]\\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\frac{p_\pi(\omega)}{p_b(\omega)}G(\omega)\right]\\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\rho(\omega)G(\omega)\right],\end{aligned} \tag{13.7}$$

where $\rho(\omega) = \frac{p_\pi(\omega)}{p_b(\omega)}$. We can expand the probability of a particular trajectory under policy μ into a product of probabilities (as this is a Markov Decision Process):

$$p_\mu(\omega) = d(S_0)\ \mu(A_0|S_0)\ p(S_1|A_0,S_0)\ \mu(A_1|S_1)\ldots\mu(A_{T-1},S_{T-1})\ p(S_T|A_{T-1},S_{T-1}). \tag{13.8}$$

Notice that for a given trajectory, the only terms unique to the policy are the ones given by $\mu(a_t|s_t)$. This means that if we find the ratio of $\frac{p_\pi(\omega)}{p_b(\omega)}$, all the terms except for the policy differences will cancel, meaning that the quantity ρ does not depend on the environment:

$$\begin{aligned}\frac{p_\pi(\omega)}{p_b(\omega)} &= \frac{\cancel{d(S_0)}\ \pi(A_0|S_0)\ \cancel{p(S_1|A_0,S_0)}\ \pi(A_1|S_1)\ldots\pi(A_{T-1},S_{T-1})\ \cancel{p(S_T|A_{T-1},S_{T-1})}}{\cancel{d(S_0)}\ b(A_0|S_0)\ \cancel{p(S_1|A_0,S_0)}\ b(A_1|S_1)\ldots b(A_{T-1},S_{T-1})\ \cancel{p(S_T|A_{T-1},S_{T-1})}}\\ &= \prod_{t=0}^{T-1}\frac{\pi(A_t|S_t)}{b(A_t|S_t)} = \rho(\omega).\end{aligned} \tag{13.9}$$

This technique of using different weights to estimate an expectation is called *importance sampling*. It has this name as it determines how to adjust the weighting of a given sample, depending on how representative it is under the other dynamics (other policy), or rather, how important it is to the expectation.

Let's take an example where $\rho(\omega) = 2$, such that ω is twice as likely to happen under π than b. In this case, it means that when we are sampling trajectories, ω will be half as likely to be sampled when we are using b, and therefore, we must double the contribution to correct for the sum.

This technique of importance sampling is extremely powerful, as it generalises how to learn from experience, regardless of the policy under which the experience was generated. In the special case that $\pi = b$, then this returns to *on-policy* learning, and all the ρ factors will be equal to 1.

There are a few caveats we must address when sampling using b. If b does not sample all the possible experiences that π samples, then there are experiences for which $p_b(\omega) = 0$, and as such, the ratio in ρ becomes undefined. The only exception to this is when $p_\pi(\omega) = 0$, in which we set $\rho(\omega) = 0$. From this, we can learn that $\pi(a|s) > 0$ implies that $b(a|s) > 0$, or in words, any possible action taken under π in a state s requires that b at least sometimes take that action. In general, we want b to be a stochastic policy in places where $\pi(a|s) > 0$.

So far, we have only come up with a way of estimating the average return of a trajectory. Let us adjust our notation so that we assess partial returns from a state s_t at time t. We will adjust ρ to be instead $\rho_{t:T}$, which is equal to:

$$\rho_{t:T} = \prod_{t'=t}^{T-1}\frac{\pi(A_{t'}|S_{t'})}{b(A_{t'}|S_{t'})}. \tag{13.10}$$

We recover our original $\rho(\omega) = \rho_{0:T}$, for the entire trajectory. The partial return of the trajectory from t until the end is given as:

$$G(\omega_{t:T}) = \sum_{t'=t}^{T-1}\gamma^{t'-t}R_{t'}. \tag{13.11}$$

With our notation adjusted, we can easily define the state-action-value function under π in terms of our behaviour policy b:

$$\begin{aligned} q_\pi(s,a) &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[G(\omega_{t:T})|S_t=s,A_t=a\right] \\ &= \mathop{\mathbb{E}}_{\omega\sim b}\left[\rho_{(t+1):T}G(\omega_{t:T})|S_t=s,A_t=a\right]. \end{aligned} \tag{13.12}$$

Under this notation, we take a trajectory ω which passes through s and taking action a at time t, and averaging the importance-sampled partial return from t. Notice how we are not including the term $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, since by definition, we chose the action A_t, in state S_t at time t, as the first action in a q function is independent of a policy. Now that we have a formula, we can modify Algorithm 2 for estimating these q values, using an *off-policy* method, as shown in Algorithm 3.

Data: Policy π, and behaviour policy b
Result: Estimated state-action-value function $Q(s,a) \approx q_\pi(s,a)$.
Initialise a state-action-value-function $Q(s,a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$;
Initialise an empty list, $Returns(s,a)$ of sampled returns for all $s \in \mathcal{S}$;
for $i \leftarrow 1$ **to** N_{max} **do**
    Sample a trajectory, ω, using b: $\omega = (S_0, A_0, R_0 \ldots, S_T, A_{T-1}, R_{T-1})$;
    $G \leftarrow 0$;
    $\rho \leftarrow 1$;
    for t *in* $T-1, T-2, \ldots, 0$ **do**
        if $t < T-1$ **then**
            $\rho \leftarrow \rho \times \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$;
        end
        Append $(R_t + \gamma\rho G)$ to $Returns(S_t, A_t)$;
        $G \leftarrow R_t + \gamma G$;
        $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$;
    end
end

Algorithm 3: State-action value function estimation of the policy π using the off-policy samples from b. This uses an every-visit Monte-Carlo algorithm.

One should note that our estimates for $q_\pi(s,a)$ are biased using the every-visit Monte-Carlo algorithm, but this bias shrinks asymptotically to zero [10] as $N_{\max} \to \infty$. One can remove this bias, using the first-visit variant, which modifies Algorithm 3 such that the importance-sampled partial return is only appended to the list of returns if the tuple (S_t, A_t) did not appear in the trajectory before t.

According to Sutton and Barto, Ref. [10], this method of estimation can have extremely high variance, and as stated in the previous paragraph, some bias. This means that one needs to sample a lot of trajectories to get a reliable estimate of the q function, meaning that *off-policy* learning can be extremely sample inefficient.

However, if one is able to get a good approximate q function, one can use policy iteration (as described in Section 12.5) to improve the policy π.

13.4 Bootstrapping: Updating estimates with estimates

Up until this point, we have estimated the value function using the Bellman equations or running trajectories and constructing an estimate via Monte Carlo methods. However, these methods are not mutually exclusive and can be synthesised into a combined method. Let's explore some of these methods in further detail.

13.4.1 Temporal Difference

We will first look at the simplest *temporal difference* method, known as $TD(0)$. Remember that we defined the value of state to be

$$v_\pi(s) = \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \tag{13.13}$$

which is just an expectation of the reward plus the discounted next state. Or, in other words:

$$v_\pi(s) = \mathop{\mathbb{E}}_{\omega\sim\pi|S_t=s}\left[R_t + \gamma v_\pi(S_{t+1})\right]. \tag{13.14}$$

This equation tells us that we could use a single transition to estimate the value of a state s. Instead of having to generate an entire trajectory, we can just jump one step into the future, and gain an estimate of the current value. We can calculate an estimate for the "error" on a value estimate with the TD error:

$$\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t). \tag{13.15}$$

An algorithm to learn the value function over time with experience is given below in Algorithm 4. In essence, the aim is to minimise the expected temporal difference error, which is minimised by the actual value function v_π. Convergence is guaranteed under stochastic conditions, such as annealing α towards 0, but ensuring that

$$\lim_{T\to\infty}\left[\alpha_T\right] = 0 \tag{13.16}$$

$$\lim_{T\to\infty}\left[\sum_{t=0}^{T}\alpha_t\right] = \infty \tag{13.17}$$

$$\lim_{T\to\infty}\left[\sum_{t=0}^{T}\alpha_t^2\right] = \text{const}, \tag{13.18}$$

which are collectively known as the **Robbins-Monro Conditions**[3]

Our algorithm is now able to update the estimate of the value function before actually finishing the trajectory, which is the real advantage of temporal difference methods as they can be used for online updates. The effectiveness of such updates depends on how good the estimate of the value function currently is. These methods are extremely helpful in non-episodic problems.

Data: Policy π, learning rate $\alpha \ll 1$ and discount γ.
Result: Estimated value function $V(s) \approx v_\pi(s)$.
Initialise a state-action-value-function $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$, except for terminal states which are set to 0.;
for $i \leftarrow 1$ **to** N_{max} **do**
  Sample an initial state s with probability s;
  while *s is not terminal* **do**
    Sample action a using π in state s;
    Sample next state with $s' \leftarrow f(s, a)$;
    Sample the reward for the action $r \leftarrow r(s', s, a)$;
    Estimate TD error $\delta \leftarrow r + \gamma V(s') - V(s)$;
    $V(s) \leftarrow V(s) + \alpha\delta$;
    Move to the next state with $s \leftarrow s'$;
  end
end

Algorithm 4: Value function estimation of the policy π by using 1 step temporal difference learning - $TD(0)$.

13.4.2 *n*-step Temporal Difference

We can extend the idea of temporal difference to instead of including a single transition, we can include a finite number of transitions. The idea is simple, we can play with the definition of the value function and "unroll" some of the rewards into a horizon, H, from the current state:

$$\begin{aligned} v_\pi(S_t) &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[R_t + \gamma R_{t+1} + \ldots \gamma^H R_{t+H} + \gamma^{H+1} v_\pi(S_{t+H+1})\right] \\ &= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\gamma^{H+1} v_\pi(S_{t+H+1}) + \sum_{t'=t}^{t+H}\gamma^{t'-t} R_{t'}\right]. \end{aligned} \tag{13.19}$$

[3] Read more about this in the Stochastic Gradient Descent section from Ref [6].

In the case of $H = 0$, we get back $TD(0)$: $V_\pi(S_t) = \mathop{\mathbb{E}}_{\omega\sim\pi}[R_t + V_\pi(S_{t+1})]$. When $H \to \infty$ or stops when hitting S_T, the terminal state, we have an equation for estimating the value function via Monte-Carlo (i.e. by running trajectories). This allows us to blend between temporal difference and Monte-Carlo methods, in which a compromise can often lead to optimal learning.

Here, we can calculate our temporal difference error to be

$$\delta_t = \sum_{t'=t}^{t+H} \gamma^{t'-t} R_{t'} + \gamma^{H+1} V_\pi(S_{t+H+1}) - V(S_t). \tag{13.20}$$

Notice that we can also sum up the temporal difference errors for an entire trajectory, regardless of H, and provided that the value estimate is not updated during the sum, we return to the Monte-Carlo "return-to-go" from the state S_t. We have therefore recovered a way of going from $TD(0)$, all the way back to a Monte-Carlo method, with every variant in-between. Remember that introduction of any amount of bootstrapping to replace a return will bias the temporal difference update, resulting in a "*semi-gradient*" method, as when we differentiate δ_t^2, we only take the derivate with respect to the current state, not the future state used to bootstrap. This bias is reduced asympotically to 0 when taking $H \to \infty$, but still exists. The bias is also removed once $V(s) \approx V_\pi(s)$ for all s.

13.5 Watkin's Q-Learning

In Q-Learning, we aim to learn what the optimal q-function is through experience. The algorithm has the following properties:

- Q-Learning is an **off-policy** method, meaning it can learn from experience gathered from another policy.
- It does not require access to transition probabilities and therefore is **model-free**.
- Has theoretical convergence to the optimal policy in infinite training time.

We start with an approximate Q function, which we aim to be equal to q^* after training. This Q approximation defines our policy

$$\pi(s) = \arg\max_{a\in\mathcal{A}(s)} Q(s,a) \quad \forall s \in \mathcal{S}. \tag{13.21}$$

Usually, we use an ϵ-greedy policy to have some exploration, denoted as

$$\pi_\epsilon(a|s) = \begin{cases} (1-\epsilon) + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \pi(s), \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq \pi(s). \end{cases} \tag{13.22}$$

As the Bellman equations for the optimal policy can be written as

$$q^*(s,a) = \mathop{\mathbb{E}}_{s',r\sim p(s',r|s,a)} \left[r + \gamma v^*(s')\right], \tag{13.23}$$

$$v^*(s) = \max_{a\in\mathcal{A}(s)} q^*(s,a), \tag{13.24}$$

we can set the target for our Q to be

$$Q(s,a) \approx \mathop{\mathbb{E}}_{s',r\sim p(s',r|s,a)} \left[r + \gamma \max_{a'\in\mathcal{A}(s')} Q(s',a')\right] \quad \forall s \in \mathcal{S}. \tag{13.25}$$

We can write a quantity $\delta(s,a)$ which estimates how far away our current Q is for a given transition from state s using action a with

$$\delta(s,a) = \mathop{\mathbb{E}}_{s',r\sim p(s',r|s,a)} \left[r + \gamma \max_{a'\in\mathcal{A}(s')} Q(s',a')\right] - Q(s,a). \tag{13.26}$$

We can write down a loss function which we aim to minimise

$$L = \frac{1}{2} \sum_s \sum_{a\in\mathcal{A}(s)} \mu(s) \left[\delta(s,a)\right]^2, \tag{13.27}$$

where $\mu(s)$ is some probability distribution over states. For now, we will not consider what this is.

In general, we can use the approach that

$$\nabla_{Q(s,a)}\delta(s,a) \approx -1, \tag{13.28}$$

which is only a semi-gradient as we do not consider the effect in the next state.

If we have a tabular Q approximation, we can update it with a learning rate α using the equation

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha\left[R_t + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a)\right], \tag{13.29}$$

which is derived from the stochastic semi-gradient descent update rule

$$Q(s,a) \leftarrow Q(s,a) - \alpha\nabla_{Q(s,a)}L, \tag{13.30}$$

where we consider the scenario that $\mu(s)$ is just some non-zero probability of being in a given state, that can be set equal to $\mu(s) = \frac{1}{|\mathcal{S}|}$. In the end, all choices of μ have the same optimum, when $\delta(s,a) = 0\ \forall s \in \mathcal{S}$.

Chapter 14

Approximate Solution Methods

Throughout the earlier RL chapters, we have been assuming that we can construct a value or state-action value function for a problem by storing a number (or many numbers) for every single state of a problem. However, this assumes that we can enumerate the states of a problem. It may be the case that there are an infinite number of states, such as in continuous domains. In these situations, one can group states such that many original states are combined into a single representative "state". This process is sometimes called *binning* when discretising a set of values as you would do when computing a histogram. As an example, imagine a robot arm with some moving parts, whose limb orientation is measured by a set of continuous signals (outputting in degrees). To reduce the state space, we can instead round all the angles to the nearest degree, *discretising* the state-space to make it smaller and more manageable.

Additionally, in extremely large state spaces, it is possible that an agent will frequently encounter states that have not been visited before. By using a simple lookup table for the value function, the agent cannot *generalise* to unseen states and infer what actions are likely to be optimal behaviour. Training these lookup tables, therefore, has to ensure that enough of the state space is covered to be useful, making the training process computationally expensive.

Take an example of the Lunar Lander[1] game, whereby an agent controls various thrusters on a spacecraft, much like the one which landed on the Moon, to bring it safely down on the surface. A game might randomise the surface of the Moon each time, along with the incoming initial speed of the lander, along with the fuel and power of the thrusters. Even in this relatively simple game, one has many continuous inputs, and a huge variety of possible states and hence an incredibly large space of possibilities, completely infeasible to solve via tabular methods. Another example is training a self-driving car. Even with extensive experience, in the real world, there will be scenarios that have not been seen before, and hence, an urgent need for the ability of the agent to generalise to new situations.

We can bring in *supervised learning* to try and learn models that can generalise to unseen situations, and possibly, learn more efficiently. SL studies various techniques to learn function approximations, based on data, that aim to describe the general function transformation from data to label, which can generalise on unseen data. Reinforcement Learning is a bit different, as it is extremely rare to have any labelled data[2] for how to act optimally in a certain situation in order to maximise a reward signal. Even with this data, it is unlikely that it is exhaustive over every possible state, and some learning may be needed to learn a generalised policy. Despite the lack of labelled data, the techniques of function approximation are still incredibly useful in the RL domain, providing the foundations for the bulk of the most recent cutting edge advances in the field. In this chapter, we will cover the basics of applying function approximation to RL problems.

14.1 Large state spaces and generalisation

As mentioned in the introduction, function approximation is the method by which we can start tying together the ideas from supervised and unsupervised learning (SL and USL respectively) with reinforce-

[1] An example can be found in the *Gym* documentation (https://www.gymlibrary.dev/environments/box2d/lunar_lander/), which is a Python module for providing many environments for use with RL algorithms.

[2] Although some methods do use examples from an expert to kickstart learning, known as *Imitation Learning*.

ment learning. The aim of SL is to find a mapping from one quantity to another which generalises well to unseen data, such as mapping images of animals to a label (i.e. cat or dog etc.). Immediately, one problem may come to mind when trying to apply this directly to RL: predicting the value or state-action-value function.

If a problem has $|\mathcal{S}|$ total states, and in each state, s, an agent can take $|\mathcal{A}(s)|$ actions, then the size of the mappings need for a tabular method is given by:

$$\text{size}(Q_\pi) = \sum_{s \in \mathcal{S}} |\mathcal{A}(s)| \leq |\mathcal{S}| \times \max_s |\mathcal{A}(s)| \tag{14.1}$$

$$\text{size}(V_\pi) = |\mathcal{S}|. \tag{14.2}$$

Even for a simple problem like grid-world, if we have 10^3 possible x and y values, and each state has at most 4 actions, then $\text{size}(Q_\pi) \leq 4 \times 10^6$ and $\text{size}(V_\pi) = 10^6$. What this means is that an algorithm which aims to solve this problem will need to keep track of approximately 4 million numbers (for the Q function), and visit all 1 million states at least once to solve the problem. Grid-world only has two dimensions currently, imagine expanding into three dimensions and this number will quickly grow out of control, and the computational and memory cost of these problems can quickly become untenable. What's more unfortunate, is that when one learns a Q function for this problem, it is only useful on the given grid-world map it was trained on, and does not generalise to other problems. Clearly these methods are extremely limited when it comes to applying them in the real world, where the majority of problems have a state space so large, and problem set so varied, that traditional tabular methods won't cut it.

Furthermore, these methods are usually unsuitable for problems with continuous state and/or action spaces. They can certainly be applied to these problems using a "binning" technique, or by approximation, but these are no longer considered strictly tabular methods, and instead, are approximate solution methods.

We can see that there are three main problems with the tabular approach:

1. Many problems have state and action spaces which are too large to be efficiently enumerated, causing a huge, infeasible, computational cost and complexity when using tabular methods.
2. Tabular methods do not generalise to unseen states and problems.
3. Tabular methods are usually unsuitable for problems with continuous state and/or action spaces.

In order to tackle these problems, we forego the idea that we can maintain a unique value estimate for each state or each state-action pair. Instead, we aim to learn a mapping from states to values (or state-action-values) - a function approximation.

14.2 Loss Functions

In order to talk about a loss function, let's introduce a simple type of function approximation: the linear function approximation, which was covered earlier in Section 2.2. To approximate the value of a state, we can combine the features of a given state linearly with associated weights. In general, this type of function looks like the following:

$$V_{\boldsymbol{\theta}}(s) = \boldsymbol{w}^T \boldsymbol{s} + b \approx v_\pi(s), \tag{14.3}$$

where we use $\boldsymbol{\theta}$ to symbolise the parameters of the model where $\boldsymbol{\theta} = [\boldsymbol{w} \;\; b]$. $\boldsymbol{w}$ is a vector of weights, with the same dimensions as $\boldsymbol{s}$, the state-vector, and b is just a constant which offsets all approximated values. Here, we are not using an activation function[3], as the predicting the value function is usually a regression problem and the function needs to be able to theoretically take any value.

We can train this function approximation by first approximating what $v_\pi(s)$ is. So far we have discussed many methods for doing this, such as Monte-Carlo returns, temporal difference or n-step returns. From this estimate, $V_\pi(s)$, we can construct an estimate for the error of our function approximation:

$$\delta = V_\pi(s) - V_{\boldsymbol{\theta}}(s), \tag{14.4}$$

[3]Technically the activation function here is the identity, the opposite of a non-linear activation function.

where our aim is to minimise the loss, δ^2, over all states. However, when we minimise the error for one specific state, say s_1, then it usually means increasing the error on a different state, s_2. For this reason, we need to specify how much attention needs to be paid to each state, or rather, the weighting applied to each part of the error. Typically, in SL, we construct a loss function which outputs a single number. This involves the sum over independent samples in your dataset. However, if you are trying to classify cats and dogs, and only 20% of your images are cats, and you just sum the errors from each image, then your optimisation will be biased towards classifying dogs correctly. This is because the dogs make up the bulk of your dataset and whose samples contribute 80% of the error towards your sum, by the nature of how often they appear. Instead, one can weight the individual importance of each sample, according to how much you want it to contribute to the overall loss. In the cats and dogs example, you can weight each cat image as $\frac{A}{20}$ and each dog image as $\frac{A}{80}$, where A is just some constant which normalises the sum of these weights to 1 (over all images). This adjustment means that a cat image will contribute $\frac{A}{20} \div \frac{A}{80} = 4$ times as much as a dog image, to compenstate for the fact that there are 4 times fewer images of cats.

In our example, how do we weigh the importance of each state when trying to get the value function approximation to match the real state-value function? One solution is to weigh the contribution of a state, by the probability that an agent will encounter it, under the current policy. We can call this probability the distribution function $\mu(s)$, which is simply the probability that an agent will be in state s, in a given episode. In the non-episodic case, we call $\mu(s)$ the *steady-state* distribution. Fortunately, we do not need to calculate this probability distribution ourselves, instead we can define our loss function as

$$L(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\omega \sim \pi} p(\omega) \sum_{t=0}^{T(\omega)-1} (V_\pi(S_t) - V_{\boldsymbol{\theta}}(S_t))^2, \tag{14.5}$$

where $p(\omega)$ is the probability of observing the trajectory ω under the policy π. Notice that we are using $V_\pi(S_t)$ here to stand in for any estimate of the value. One choice could be the return-to-go from state S_t, denoted as G_t. Since this is a weighted sum of probabilities, this just becomes the expected total squared error of a trajectory:

$$L(\boldsymbol{\theta}) = \frac{1}{2} \mathop{\mathbb{E}}_{\omega \sim \pi} \left[\sum_{t=0}^{T(\omega)-1} (V_\pi(S_t) - V_{\boldsymbol{\theta}}(S_t))^2 \right], \tag{14.6}$$

which can be further simplified into

$$\begin{aligned} L(\boldsymbol{\theta}) &= \frac{1}{2} \sum_{s \in \mathcal{S}} \mu(s) (V_\pi(s) - V_{\boldsymbol{\theta}}(s))^2 \\ &= \frac{1}{2} \mathop{\mathbb{E}}_{s \sim \pi} \left[(V_\pi(s) - V_{\boldsymbol{\theta}}(s))^2 \right] \\ &= \frac{1}{2} \mathop{\mathbb{E}}_{s \sim \pi} \left[\delta(s)^2 \right], \end{aligned} \tag{14.7}$$

where $\delta(s)$ is the error of the function approximation compared to the real value function at state s. For context, this defines $\mu(s)$ as

$$\mu(s) = d(s') + \sum_{s' \in \mathcal{S}} \mu(s') \sum_{a \in \mathcal{A}(s')} \pi(a|s') p(s|s', a), \tag{14.8}$$

where $d(s')$ is the probability that a trajectory starts in state s'. We normalise $\mu(s)$ such that

$$\sum_{s \in \mathcal{S}} \mu(s) = \mathop{\mathbb{E}}_{\omega \sim \pi} |T(\omega) - 1|, \tag{14.9}$$

which means that $\mu(s)$ is normalised such that the sum over all states is equal to the expected number of transitions in trajectory. One could also modify the loss function so that the sum would be set to 1, but this would scale the loss function, which is arbitrary and does not affect the minimum.

One should note that we do not ever need to calculate $\mu(s)$ directly, but when we are sampling experience in an on-policy fashion, we need only sum up contributions to the total error from the states actually visited.

14.2.1 Off-Policy Alteration

We can still learn a value function using the same techniques from the previous chapter, but by sampling states according to a behaviour policy b, and then correcting each term in the sum with a factor of $\frac{\mu_\pi(s)}{\mu_b(s)}$. The derviation is left out of the notes here, but the techniques are almost identical to the derivation in Section 13.3.

14.3 Stochastic Gradient Descent

If our function approximation is differentiable, such as a neural network, we can simply take the derivative of our loss function, with respect to the parameters of the network to update them:

$$\boldsymbol{\theta} \to \boldsymbol{\theta} - \alpha \frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}}, \tag{14.10}$$

where α is just a step size parameter, much like a learning rate. We can calculate the derivative of our loss function with respect to the value function, assuming that π is independent of our approximate value function $V_{\boldsymbol{\theta}}$:

$$\begin{aligned}
\frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}} &= \frac{1}{2}\nabla_\theta \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t\in N(\omega)} (V_\pi(s_t) - V_{\boldsymbol{\theta}}(s_t))^2\right] \\
&= \frac{1}{2}\mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t\in N(\omega)} \nabla_\theta (V_\pi(s_t) - V_{\boldsymbol{\theta}}(s_t))^2\right] \\
&= \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t\in N(\omega)} (V_\pi(s_t) - V_{\boldsymbol{\theta}}(s_t)) \frac{dV_{\boldsymbol{\theta}}(s)}{d\theta}\Big|_{s=s_t}\right]
\end{aligned} \tag{14.11}$$

As this is just an expectation over the individual gradients, we can just sample on policy trajectories and average the gradients of each state, weighted by the error of that state. We can apply a similar derivation to eq. (14.7) which results in the following gradient:

$$\frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}} = \mathop{\mathbb{E}}_{s\sim\pi}\left[\delta(s)\frac{dV_{\boldsymbol{\theta}}(s)}{d\theta}\right]. \tag{14.12}$$

If we want to extend this to a stochastic gradient descent method, we can simply sample a state (under the current policy π) and evaluate the gradient. Usually this is done in mini-batches, to construct a more accurate estimate of the gradient, so a larger step size can be used.

Earlier, we stated that we should use an estimate for the error of a state $\delta(s)$, using any of the policy evaluation techniques previously discussed. However, one should take care, as if you are using a bootstrapping method, which estimates the value of a state using the estimate of a value at a future state, the estimate of the error gained is biased. More importantly, this estimate is not independent of the states sampled to construct the gradient, introducing further bias to the estimate, leading to potential instability. Using Monte-Carlo returns for the estimates is the only way that is unbiased here (i.e. setting $H = \infty$ in n-step temporal difference). Additionally, in our derivation, we assume that $V_\pi(s)$ does not depend on $\boldsymbol{\theta}$, however, if we use a bootstrapping method, this assumption is no longer true, as therefore, this method is no longer a true-gradient method, but rather, a *semi-gradient method*. Often enough, these methods still work well enough to train models, but have very few convergence guarantees, even in the simplest cases.

14.4 The Deadly Triad

Sutton and Barto, Ref. [10], warn of the danger of instability and divergence that arises whenever one combines any of the three elements in RL:

1. **Function approximation**: A scalable and efficient way of generalising a function on a problem with a large state or action space.

2. **Bootstrapping**: An efficient way of updating target values using existing estimates (as in dynamic programming, n-step returns or temporal difference methods).
3. **Off-policy training**: Training on a distribution of transitions other than that produced by the target policy.

Each of these methods have their place, and can often be an essential part of successfully training an RL agent on difficult problems. However, they are a recipe for instability. Most of the cutting edge research into reinforcement learning focuses on heuristics and techniques to help reduce the instability caused by using or combining these methods.

Sutton and Barto, Ref. [10], refrain from commenting on specific solutions, as all are open research questions, and there are no clear strategies which can be used to improve training. However, there are a few techniques that are used commonly, such as replay buffers, target value functions, asynchronous updates, policy gradients and planning. As with SL or USL, increasing the amount of data one has access to can also help mitigate some of these problems, which is achieved by efficiently simulating the reinforcement learning environment. Simulating your environments can help you scale up and parallelise the experience gathering process to get more accurate estimates of the quantities described in this text.

14.5 Summary

We are able to use any of the techniques presented in Chapter 12 and Chapter 13 to construct estimates for what the value at a given state should be. These estimates can then be used to train a custom model to predict the correct value function (or even state-action value function[4]). Note that temporal difference methods can often give a huge computational advantage over Monte-Carlo methods, but their introduction often causes bias in the estimates, leading to poor convergence guarantees. In most applications, this is an acceptable trade-off, but must be considered when implementing these methods.

When constructing a model to approximate a value function, you can mix and match any of the techniques covered in Part I and Part II, or the in the field of machine learning more generally. The methods here give you a way to gather some labelled data to learn from, given the data was generated through experience. While value and state-action-value function approximations are powerful techniques, they have poor convergence guarantees and do not offer a good way of extending to continuous action spaces effectively. Additionally, for some problems, the form that the value function takes can be extremely complex and difficult to learn, while the policy itself can be rather simple and easier to learn. For these reasons, it is very popular to directly approximate the policy function instead, without the need of a value function. However, using value function approximation can aid the stability of directly training a policy approximation. Policy gradient methods will be the topic of the next chapter.

[4]Search for "Watkins Q-Learning" for a simple algorithm to read more about these methods.

Chapter 15

Policy Gradient Methods

Up until now, we have discussed evaluating a policy by calculating the associated value function, and then improving on that policy. This focussed entirely on the value function and the estimates. However, some problems can have extremely complicated value functions, but relatively simple policies. If we only care about the value function in so far as it helps us find an optimal policy, why not just try and directly find the optimal policy.

Remember that we started with a value function as we could guarantee convergence to the optimal policy through policy evaluation and iteration. However, as we started to move to more complicated methods, introducing function approximation, off-policy learning and bootstrapping techniques, we relaxed any guarantees about optimality that we started with. For the most difficult problems, our aim is only to *attempt* to maximise the expected reward on a given problem, or get as close as possible, without guaranteeing the optimal behaviour.

Our relaxed constraint enables us to devise methods that do not involve calculating the value function, but instead, allow us to directly optimise a policy. From the title of this chapter "Policy *Gradient* Methods", we are implying that we will be discussing methods which have a differentiable, parameterised policy, such as a function approximation. These techniques are incredibly powerful and can be extended to problems in continuous state and action spaces. However, we will usually only be able to locally optimise the function (exploitation), and we often have to add in additional techniques to ensure exploration.

15.1 Parameterising a policy

Up until now, our policy has been fairly simple, relying on a value or state-action-value function to choose the actions. However, we can go directly from a state to a policy. The typical way of creating this mapping is by using a neural network as, in theory, it can represent any function due to its *universal-approximation* trait. NNs are also useful in that they are well studied in supervised learning, and there are many open source libraries which provide ways of constructing and optimising them.

We can remember that a neural network can be thought of as a collection of weights, biases and activation functions. We consider only a feed-forward neural network here, which can be written as:

$$\boldsymbol{x}_i = \sigma_i(W_i^T \boldsymbol{x}_{i-1} + \boldsymbol{b}_i), \tag{15.1}$$

where σ_i represents the activation function, $\boldsymbol{x}_i$ the output, W_i and $\boldsymbol{b}_i$ the weights and biases respectively of the i^{th} layer. Note that $\boldsymbol{x}_0$ is the input to the neural network.

15.1.1 Discrete Action Spaces

If we are trying to represent a policy for an environment which accepts only deterministic actions, then we should ensure the final layer of the neural network is able to output a probability distribution of selecting the available actions. We can have the neural network output a preference for each action, and the relative preferences decide the final probability distribution. In neural networks, we call these preferences *logits*. If a neural network has H hidden layers (not including the input or final output layer),

then the H^{th} layer should have a linear activation function, with $(\max_s |\mathcal{A}(s)|)$ outputs (the maximum number of possible actions).

The final layer, $H+1$, should have a *softmax* activation function. The softmax function can be written as:

$$x_{H+1}^{(j)} = \frac{\exp(x_H^{(j)})}{Z}, \tag{15.2}$$

where $x^{(j)}$ represents the j^{th} component of the vector $\boldsymbol{x}$ and Z is the term that normalises the output distribution, so it can be interpreted as a probability distribution:

$$\begin{aligned} 1 &= \sum_j x_{H+1}^{(j)} \\ 1 &= \sum_j \frac{\exp(x_H^{(j)})}{Z} \\ Z &= \sum_j \exp(x_H^{(j)}), \end{aligned} \tag{15.3}$$

which gives us a value for Z, often called the *partition-sum* in Physics. The only difference is in the binary case (only two possible actions), we need only a single preference value to indicate the probability. We can use the *logistic sigmoid* which is represented by

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{15.4}$$

which has the properties that $\sigma(-x) = (1-\sigma(x))$. If $x = 0$, i.e. the preference is 0, then there is an equal chance of choosing either action. Notice that when $x \to \infty$ then $\sigma(x) \to 1$ and when $x \to -\infty$ then $\sigma(x) \to 0$. This bounds the probabilities between 0 and 1.

Let's think about why we have chosen functions which make our policy probabilistic, surely it would be easier to simply choose the maximum as a preference and have a deterministic policy? In the case of policy gradients, this is a harder choice. If we choose a soft policy, where the probability of each action is strictly non-zero, then we have already built in some exploration, so that we can gain information about the effect of different actions. A soft policy allows us to focus on *on-policy* methods for optimisation. However, unlike the ϵ-*greedy* policies as before, we can adapt the exploration over time, to stop exploring parts of the policy space which are known to be bad.

We can additionally use an off-policy method (such as ϵ-*greedy*) with the appropriate importance sampling on any expectation value over the trajectories. Since we have a soft policy in both cases, we will always get a well-defined importance sampling ratio, leading to better stability, while also providing exploration if needed during training.

15.1.2 Continuous Action Spaces

As we are directly parameterising the policy, we introduce the idea of continuous actions. These are common in practice, e.g. a robot trying to walk needs to tell each of the servo motors the angle that it should move towards. As before, one can discretise the space of continuous actions and use the method in the previous section, however, we are not limited by this method.

One method for directly choosing continuous actions is for the network to output the mean and variance of a Gaussian probability distribution which is sampled to choose the actual continuous action. The reason that we make the network output a probability distribution is that we can efficiently differentiate this probability distribution to update the weights of a network, along with introducing the exploration that comes hand-in-hand with a soft policy.

As a concrete example, let's imagine a linear network for a policy, which chooses the values for a Gaussian probability distribution, which in turn samples the continuous actions for the policy. We can then see that the probability density of the Gaussian policy, π, is:

$$\pi_\theta(a|s) = \frac{1}{\sigma_\theta(s)\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(a-\mu_\theta(s))^2}{\sigma_\theta(s)^2}\right), \tag{15.5}$$

where $\mu_\theta(s)$ and $\sigma_\theta(s)$ are the parameterised mean and standard deviation of the Gaussian probability density function which describes the policy in the state s. As before, we represent the parameters of the model with θ.

In a continuous action space, it is known that the probability of selection exactly an action, a, is 0 for any action a. Instead, we can ask what is the probability that an action being chosen between two limits a_1 and a_2, such that $a_1 \leq a \leq a_2$, which is given by

$$\Pr\left[a_1 \leq a \leq a_2 | s\right] = \int_{a_1}^{a_2} \pi(a|s) da. \tag{15.6}$$

For our purposes, we do not need to intimately understand this function, but instead, be able to sample and differentiate it.

Sampling a Gaussian Distribution

In code, we can sample a Gaussian distribution with mean μ and standard deviation σ if we have access to a function that can generate a normally distributed random number with mean 0 and standard deviation of 1. The standard notation for a Gaussian PDF[1] with mean μ and standard deviation σ is given by $\mathcal{N}(\mu, \sigma^2)$, where σ^2 is the *variance* of the PDF. We can adjust a unitary Gaussian to our distribution through the following transformation:

$$r' = \mu + r\sigma, \tag{15.7}$$

where r is distributed according to $\mathcal{N}(0, 1^2)$ and r' is distributed according to $\mathcal{N}(\mu, \sigma^2)$. Most programming languages will have a way of generating normally distributed unitary numbers, for example in the Python programming language as shown in Listing 15.1.

```python
import random
def gauss_rand(mu, sigma):
    r = random.gauss() # Default zero mean and unit variance
    return (mu + r * sigma)
```

Listing 15.1: Generating Gaussian random numbers in Python

The function we used in Python actually allows us to change the mean and standard deviation directly in the function with optional arguments, but this is not always available in other languages, hence why it is shown here.

Differentiating a Gaussian

We can differentiate the policy w.r.t to the parameterised mean μ_θ as the following:

$$\frac{\partial \pi_\theta(a|s)}{\partial \mu_\theta(s)} = \frac{a - \mu_\theta(s)}{\sigma_\theta(s)^2} \pi_\theta(a|s). \tag{15.8}$$

We can write a similar equation for the variance, with a little more involved derivation:

$$\frac{\partial \pi_\theta(a|s)}{\partial \sigma_\theta(s)} = \frac{(a - \mu_\theta(s))^2 - \sigma_\theta(s)^2}{\sigma_\theta(s)^3} \pi_\theta(a|s). \tag{15.9}$$

Notice that we only calculated the partial derivatives. In general, our full derivative, w.r.t the parameters θ, is given by:

$$\frac{d\pi_\theta(a|s)}{d\theta} = \frac{\partial \pi_\theta(a|s)}{\partial \mu_\theta(s)} \frac{d\mu_\theta(s)}{d\theta} + \frac{\partial \pi_\theta(a|s)}{\partial \sigma_\theta(s)} \frac{d\sigma_\theta(s)}{d\theta}. \tag{15.10}$$

This may look complicated, however, only equations (15.8) and (15.9) are manually used to calculate the derivative, the other part of the derivative can be calculated using an automatic differentiation library, common in all deep learning packages like *Pytorch* or *Tensorflow*.

[1] Probability density function.

15.2 Policy Gradient: REINFORCE

Now that we have discussed ways to parameterise a policy, such that it is differentiable, let's write down a loss function. The most sensible loss function to optimise is tied to the expected return. In the episodic case, we can define this as

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t=0}^{T(\omega)} \gamma^t R_t\right], \tag{15.11}$$

where $T(\omega)$ is the length of the trajectory ω. However, in this form, it is unclear how the policy affects this value, however, one can instead write:

$$\mathcal{L}(\theta) = \sum_{\omega} p(\omega)\left[\sum_{t=0}^{T(\omega)-1} \gamma^t R_t\right], \tag{15.12}$$

wherein $p(\omega)$ can be expanded into

$$p(\omega) = d(S_0)\prod_{t=0}^{T(\omega)-1} \pi_\theta(A_t|S_t)p(S_{t+1}, R_t|A_t, S_t) \tag{15.13}$$

as usual. Notice here that $p(\omega)$ explicitly depends on θ, but only in the terms $\pi_\theta(A_t|S_t)$ and no other terms.

Let's derive the derivative of our loss function w.r.t to the parameters θ:

$$\begin{aligned}\nabla_\theta\mathcal{L}(\theta) &= \sum_{\omega}\nabla_\theta\left[p(\omega)\sum_{t=0}^{T(\omega)-1}\gamma^t R_t\right]\\ &= \sum_{\omega}\left[\sum_{t=0}^{T(\omega)-1}\gamma^t R_t\right]\nabla_\theta p(\omega).\end{aligned} \tag{15.14}$$

Here, we are using the notation ∇_θ instead of $\frac{d}{d\theta}$, as θ is usually a vector of parameters. You can think of ∇_θ as a vector with components $[\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \ldots, \frac{\partial}{\partial\theta_n}]$, where n is the number of parameters.

We know that the differential of the total discounted reward is not dependent on θ as these are just numbers, the only dependence is on $p(\omega)$. We can explicitly work out the derivative of this probability:

$$\begin{aligned}\nabla_\theta p(\omega) &= \nabla_\theta d(S_0)\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t)p(S_{t+1}, R_t|A_t, S_t)\\ &= \left[d(S_0)\prod_{t'=0}^{T(\omega)-1} p(S_{t'+1}, R_{t'}|A_{t'}, S_{t'})\right]\nabla_\theta\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t),\end{aligned} \tag{15.15}$$

where we have moved all terms independent of θ to the left, so we can focus on the differential. Here, we use the product rule on the product of probabilities:

$$\nabla_\theta\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t) = \left[\prod_{t''=0}^{T(\omega)-1}\pi_\theta(A_{t''}|S_{t''})\right]\sum_{t=0}^{T(\omega)-1}\frac{1}{\pi_\theta(A_t|S_t)}\nabla_\theta\pi_\theta(A_t|S_t). \tag{15.16}$$

We can use the fact that

$$\nabla_\theta\log\pi_\theta(a|s) = \frac{1}{\pi_\theta(a|s)}\nabla_\theta\pi_\theta(a|s) \tag{15.17}$$

to simplify the equation:

$$\nabla_\theta\prod_{t=0}^{T(\omega)-1}\pi_\theta(A_t|S_t) = \left[\prod_{t''=0}^{T(\omega)-1}\pi_\theta(A_{t''}|S_{t''})\right]\sum_{t=0}^{T(\omega)-1}\nabla_\theta\log\pi_\theta(a|s). \tag{15.18}$$

Fortunately, we can substitute this all back into eq. (15.15), combining all product terms to recover $p(\omega)$ on the front:

$$\nabla_\theta p(\omega) = p(\omega) \sum_{t=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{15.19}$$

Finally, this can be put back into eq. (15.14) to get:

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) &= \sum_\omega p(\omega) \left[\sum_{t=0}^{T(\omega)-1} \gamma^t R_t \right] \times \left[\sum_{t'=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right] \\
&= \mathop{\mathbb{E}}_{\omega\sim\pi} \left[\left(\sum_{t=0}^{T(\omega)-1} \gamma^t R_t \right) \times \left(\sum_{t'=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_{t'}|s_{t'}) \right) \right] \\
&= \mathop{\mathbb{E}}_{\omega\sim\pi} \left[G_0(\omega) \times \left(\sum_{t=0}^{T(\omega)-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \right],
\end{aligned} \tag{15.20}$$

where $G_0(\omega)$ represents the total discounted return from the initial state. Notice that the gradient is defined using expectation value over trajectories. This makes an estimate easy to construct by just sampling trajectories and averaging the gradient contributions from each.

In deep learning libraries which provide automatic differentiation capabilities, one usually represents the loss function as

$$L(\theta) = \sum_{\omega\sim\pi} A(\omega) \sum_{S_t, A_t \in \omega} \log \pi_\theta(a_t|s_t), \tag{15.21}$$

where $A(\omega)$ has all gradient calculations "turned-off", and is treated just as a number. If $A(\omega)$ depends on θ explictly, then we will be using a *semi-gradient* method for optimisation, which introduces some bias to our gradients. As we will learn in later sections, this number $A(\omega)$ can be constructed in many ways. This loss function is quantitatively different from the total expected reward, however, it has the same gradient which can be used to update the weights θ. One usually estimates the gradient using a *mini-batch*, consisting of a small number of trajectories.

The most straightforward way of improving a parameterised policy, now that we have a way of estimating the gradient, is to perform stochastic gradient ascent, where the updates of the parameters will look like

$$\theta \leftarrow \theta + \alpha \frac{dL(\theta)}{d\theta}\bigg|_\theta, \tag{15.22}$$

where α is just a step-size parameter (or "learning rate") and our gradient is estimated for our parameters θ. For clarity, we have included an example of some Python-like pseudocode, which uses *Pytorch* to simulate and run a Monte-Carlo policy gradient algorithm in Listing 15.2. This algorithm is quite long as it does both Monte-Carlo sampling, and estimates the gradient to train the algorithm. The inputs are the `policy`, which is the parameterised policy π_θ, along with the environment, `env`, which represents the object which can step the environment and calculate the rewards for a given action. We also use an "optimiser" (`optimiser`), which is linked to the parameters of the policy and provides ways of updating the gradients. In the simplest case, the optimiser uses the gradients to update via Equation (15.22), but can also use a more sophisticated update rule like ADAM [5].

```
import torch # Name of Pytorch package
def step_policy(policy, env, optimiser, batch_size):
    optimiser.zero_grad() # Zero-out gradients for network parameters
    for i in range(batch_size):
        env.reset()
        states = []
        actions = []
        rewards = []
        while not env.has_finished():
            state = env.observe()
            prob_actions = policy(state) # One hot actions
            action = torch.sample(prob_actions)
            reward = env.step(action)
            states.append(state)
            actions.append(action)
```

```
            rewards.append(reward)
        G = total_discounted_reward(rewards)
        total_loss = 0
        for (state, action) in zip(states, actions):
            # One-hot encode to extract the probability of action
            one_hot_action = one_hot(action, env.num_actions)
            prob_action = one_hot_action.T * policy(state)
            total_loss += G * torch.log(prob_action)
        total_loss = total_loss / batch_size # Average over batches
        total_loss.backward() # Accumulate gradients with back-prop
    # Update the parameters of the network
    optimiser.step()
```

Listing 15.2: Python pseudocode for REINFORCE using Pytorch

We should be clear in that using a Monte-Carlo method like this requires that we run an entire trajectory before we can calculate the gradients as we need to multiply by the entire return for a trajectory.

15.3 Variance of the Policy Gradient

In policy gradient methods, we often talk about methods to "reduce the variance" of the gradient estimate. One estimate the true gradient using

$$\nabla_\theta \mathcal{L}(\theta) \approx \nabla_\theta L(\theta) = \frac{1}{N} \sum_i G(\omega^{(i)}) \left(\sum_{t=0}^{T(\omega_i)-1} \nabla_\theta \log \pi_\theta(A_t^{(i)} | S_t^{(i)}) \right), \tag{15.23}$$

where each trajectory, $\omega^{(i)}$, is sampled using π with N total samples. We can also use a single trajectory to try and estimate the gradient. The estimate of the gradient using one trajectory ω can be written as

$$\delta L = \sum_{t=0}^{T(\omega)-1} G(\omega) \nabla_\theta \log \pi_\theta(A_t | S_t). \tag{15.24}$$

As our estimate in eq. (15.23) is just a mean over other δL, we know that the variance should follow

$$\text{Var}(\nabla_\theta L) = \frac{\text{Var}(\delta L)}{N}, \tag{15.25}$$

where $\text{Var}(X)$ denotes the variance of the random variable X. We can calculate the variance with the formula

$$\text{Var}(\delta L) = \mathop{\mathbb{E}}_{\omega \sim \pi} \left[(\delta L)^2 \right] - \mathop{\mathbb{E}}_{\omega \sim \pi} \left[(\delta L) \right]^2. \tag{15.26}$$

However, we can just as easily sample a set of δL and plot a histogram to visually show the variance in the form of a wider distribution.

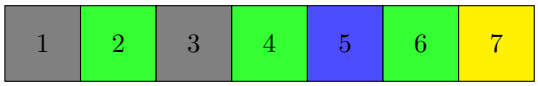

Figure 15.1: A simple grid-world environment in which the agent always starts in the state (1) in the upper-left hand corner.

Let's look at a simple grid-world environment with the map shown in Figure 15.1, which has the agent starting in the state (1) every time, and can only move east or west. Additionally, we only let the agent walk for a maximum 20 steps and set $\gamma = 1$. Note that enforcing a maximum time step of 20 introduces time as a factor in the state, however, we will only take this into account when calculating the true value functions.

15.3.1 Linear Model

We can construct a policy approximation by simple using a linear function approximation. We can encode the input state as $\tilde{\boldsymbol{s}} = [x, 1]$, which is a column vector with size 2 by 1. Appending the constant

1 to the state allows us to encode a bias vector directly into the weight matrix. We encode the weight vector as $W_\theta = [\theta_1, \theta_2]$, such that

$$\pi_\theta(\rightarrow |s) = \sigma({W_\theta}^T \tilde{\boldsymbol{s}}) \tag{15.27}$$

$$\pi_\theta(\leftarrow |s) = \sigma(-{W_\theta}^T \tilde{\boldsymbol{s}}). \tag{15.28}$$

This ensures that the probabilities are normalised as $1 - \sigma(x) = \sigma(-x)$. If we encode the actions of $(\leftarrow, \rightarrow)$ to be $(-1, +1)$ respectively, such that the environment update is just $f(x, a) = x + a$. Also, when we are at the left-most side, the probability is deterministic, always going to the right, which will have 0 gradient.

Remember that we would like to be able to calculate the gradient $\log \pi_\theta(a|s)$ which can be derived as follows:

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta \log \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= \frac{1}{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})} \nabla_\theta \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= \frac{1}{\cancel{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})}} (1 - \sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})) \cancel{\sigma(a{W_\theta}^T \tilde{\boldsymbol{s}})} \nabla_\theta (a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= (a\tilde{\boldsymbol{s}}) \times \sigma(-a{W_\theta}^T \tilde{\boldsymbol{s}}) \\
&= a\pi_\theta(-a|s)\tilde{\boldsymbol{s}}.
\end{aligned} \tag{15.29}$$

15.3.2 Variance of Linear Model

We can calculate an estimate for the REINFORCE gradient, substituting eq. (15.29) in for $\nabla_\theta \log \pi_\theta(a|s)$ to get an expression for the gradient of the loss:

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta) &= \mathop{\mathbb{E}}_{\omega \sim \pi} [\nabla_\theta l(\omega)] \\
&= \mathop{\mathbb{E}}_{\omega \sim \pi} \left[G_0(\omega) \sum_{t=0}^{T(\omega)-1} A_t \pi_\theta(-A_t|S_t) \begin{bmatrix} X_t \\ 1 \end{bmatrix} \right].
\end{aligned} \tag{15.30}$$

Now that we have an equation for calculating the gradient, we can simply run some trajectories and sample values for the gradients, which are both random variables. We can see their distribution by using plotting the approximate probability density function[2] of the random variables, as shown in Figure 15.2.

We can see that if we take only a single example, we are very unlikely to be close to the actual mean. We can only obtain an estimate close to mean by averaging many independent samples. Why is this important? Having an accurate gradient allows us to take fewer steps to converge to a locally optimal policy as we can safely increase the step-size. Techniques that require few samples to determine the mean value are known as "low-variance". Low variance techniques are efficient as one can sample fewer trajectories at each update, or choose to make larger changes to the parameters as the gradient estimate will be more accurate. In the next few sections, we will discuss ways of improving our estimate of the gradient by lowering the variance.

15.3.3 Removing the Past

In our basic estimate of the gradient in eq. (15.20), we notice that each term in the sum is multiplied by the same prefactor, $G_0(\omega)$. The other part of the term inside the sum over t is the $\nabla_\theta \log \pi_\theta(A_t|S_t)$ part, which can be interpreted as moving in the direction of increasing the likelihood of choosing the action A_t in the state S_t if multiplied by a positive number. However, we formulate all of our problems as Markov Decision Processes, which means that the history of a trajectory is not relevant to the future, only the current state is. Since we are dealing with MDPs, it does not make sense that the prefactor for each of the terms should include information from the *past*, before state S_t was reached. Fortunately, our estimate of the gradient is invariant under a special transformation, allowing us to remove the past.

[2] This can be interpreted as a type of "smooth" histogram.

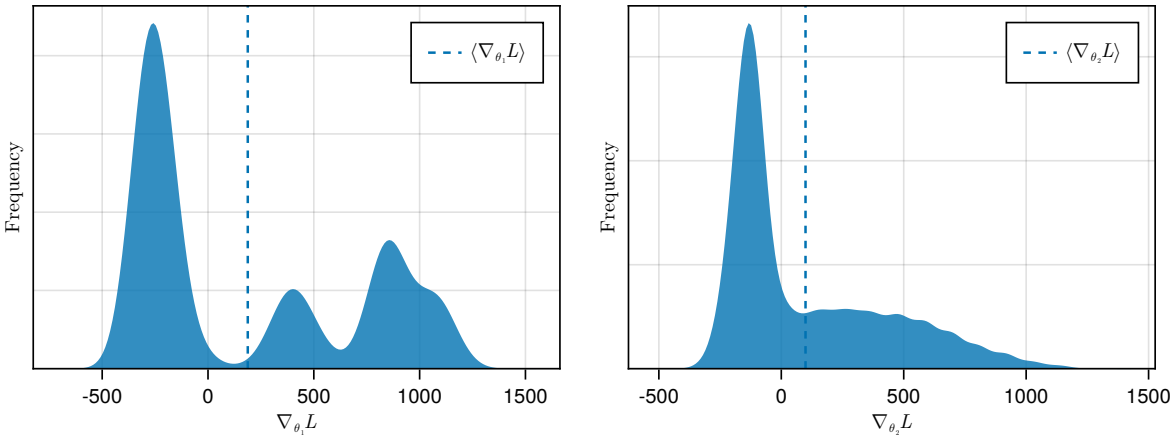

Figure 15.2: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right). Both estimates will have a very high variance, as seen by the wide distributions, however, the means are accurate and unbiased. Averaging enough of the estimates will lead to an accurate, unbiased, estimate of the true gradient. The PDFs (Probability Density Functions) are estimate by sampling $10,000$ trajectories on the excursion shown in Figure 15.1.

Take the expectation over $a \in \mathcal{A}(s)$ of the quantity $\beta(s)\nabla_\theta \log \pi_\theta(a|s)$, for any function β that only depends on the state s:

$$\begin{aligned}
\sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s)\beta(s)\nabla_\theta \log \pi_\theta(a|s) &= \beta(s) \sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s) \\
&= \beta(s) \sum_{a\in\mathcal{A}(s)} \cancel{\pi_\theta(a|s)} \frac{1}{\cancel{\pi_\theta(a|s)}} \nabla_\theta \pi_\theta(a|s) \\
&= \beta(s) \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi_\theta(a|s) \\
&= \beta(s)\nabla_\theta \sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s) \\
&= \beta(s)\nabla_\theta 1 \\
&= 0, \qquad (15.31)
\end{aligned}$$

where we used the fact that $\pi_\theta(a|s)$ is normalised to 1, a constant, whose gradient is 0. We can interpret this fact by saying that whenever we do an expectation over the gradient of the log probabilities, multiplied by a function which does not depend on the action a, but only on the state s, the resulting expectation is zero. Said differently, our expectation value is *invariant* under additions of arbitrary functions $\beta(s)$ next to $\nabla_\theta \log \pi_\theta(a|s)$. We can write this new gradient estimate as:

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi} \left[\sum_{t=0}^{T(\omega)-1} [G_0(\omega) - \beta(s)] \nabla_\theta \log \pi_\theta(a_t|s_t) \right]. \qquad (15.32)$$

This form of the estimate is commonly called the baseline, and there are many forms which the function β can take. $\beta(s)$ can even depend on the previous part of the trajectory, before t, without affecting the expected value. One popular choice for our baseline function is

$$\beta(s,t)_{\text{past}} = \sum_{t'=0}^{t-1} \gamma^{t'} R_{t'}, \qquad (15.33)$$

which is simply just the first part of the discounted reward totalled up to just before time t. This turns our prefactor into simply $\sum_{t'=t}^{T(\omega)-1} \gamma^{t'} R_{t'}$, which is the *return-to-go* from state S_t, multiplied by a discount factor of γ^t.

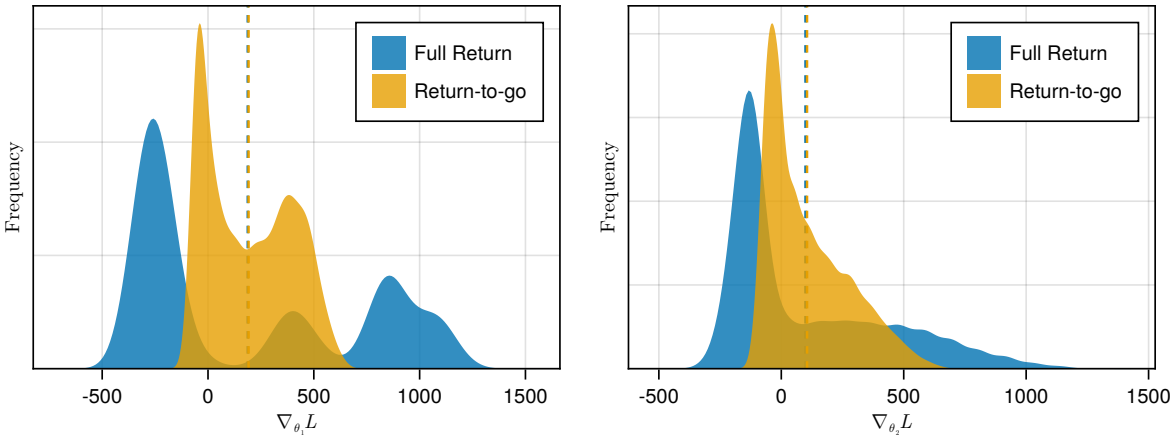

Figure 15.3: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right) compared between using eq. (15.20), shown in blue, and eq. (15.34), shown in orange. Removing the past from the rewards changes the distribution of the gradients for each parameter, but does not change the mean of the distribution. Additionally, the variance is much reduced by removing the past, allowing one to obtain a more accurate gradient estimate with fewer trajectory samples.

Now, we have removed the past from our equation, getting an estimate using samples from

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t=0}^{T(\omega)-1}\left(\sum_{t'=t}^{T(\omega)-1}\gamma^{t'}R_{t'}\right)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{15.34}$$

By definition, removing the past like this leaves the true mean unaffected, however, it makes a huge difference to the variance of the estimate. Looking at Figure 15.3, we can see that this alteration significantly reduces the variance of our estimate of the gradient for both parameters, while leaving the mean unchanged. Again, this means that we can construct an accurate (and unbiased) estimate for the gradient with very few trajectories.

In the next few sections, we will constantly come back to this idea of reducing the variance of our gradient estimate, as it is the most important thing we can do to increase the efficiency of our training with policy gradient methods.

15.4 REINFORCE with Value Baseline

From this point forward, we will take as a given that we will use eq. (15.34) (removing the past) as our starting point. To introduce some additional notation, we will introduce the concept of the *advantage*, which is usually denoted as A, which is the similar to our notation for the action A_t. For this reason, we instead denote the advantage at time t in the trajectory ω with the symbol χ_t. We now write our gradient estimate as follows:

$$\nabla_\theta \mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\sum_{t=0}^{T(\omega)-1}\gamma^t \chi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{15.35}$$

Our standard algorithm sets $\chi_t = G_t$, where G_t is the *return-to-go* for the trajectory ω, which is defined as:

$$G_t = \sum_{t'=t}^{T(\omega)-1}\gamma^{t'-t}R_{t'}. \tag{15.36}$$

However, we know that we are free to offset χ_t by any quantity that only depends on the current state, or that occur before t, without changing the estimate. Another popular choice for this quantity comes from subtracting the value of the state S_t under the current policy from G_t:

$$\chi_t = G_t - v_\pi(S_t), \tag{15.37}$$

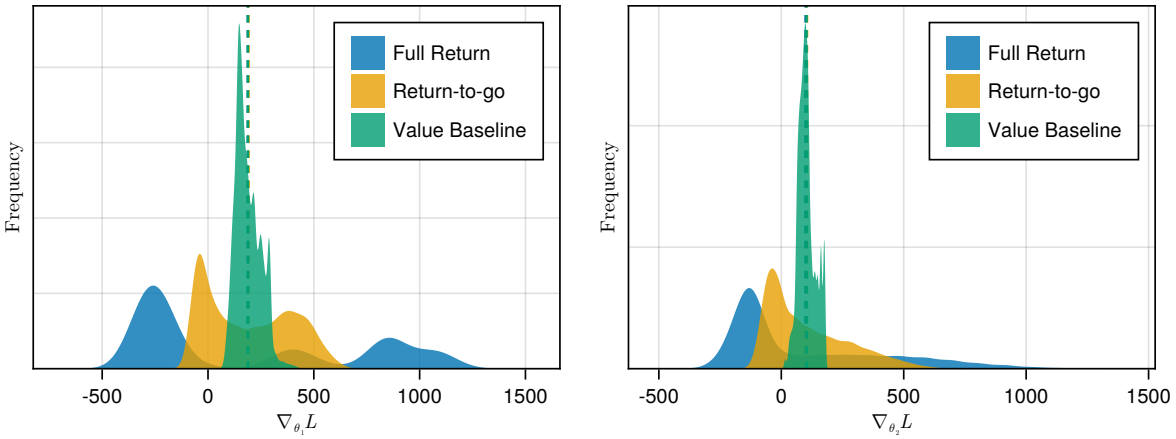

Figure 15.4: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right) compared between using eq. (15.20), shown in blue, and eq. (15.34), shown in orange. In green, we see the effect of using a value baseline using eq. (15.35) with eq. (15.37). Adding a value baseline on top of removing the history significantly reduces the variance of the estimates, but the mean is largely unaffected. Value function was estimated using 10^5 trajectories for illustration purposes.

which is where the name *advantage* comes from: since the value $v_\pi(S_t)$ is the average of G_t from that state, this quantity measures the advantage of taking the actions that let to G_t when compared with the average behaviour. The only issue here is that $v_\pi(S_t)$ is the value function of our policy, which is often unknown. However, using an estimate V_π instead will not bias your estimate, but may increase the variance if V_π is inaccurate. However, one can find that the more accurate V_π, the better it is at reducing the variance of your estimate.

15.5 Actor-Critic

In the previous methods, we still have to generate entire trajectories, which can take a very long time. Instead, it may be beneficial to be able to have a method that can learn from single transitions, so the methods can be effectively ported to non-episodic problems. Instead of generating an entire trajectory, we can instead sample single transitions and estimate the temporal difference in value between the two states. We can substitute our expression for G_t with a single step evaluation $(R_t + v_\pi(S_t))$ to get:

$$\chi_t = R_t + \gamma v_\pi(S_t) - v_\pi(S_t), \tag{15.38}$$

which we call the *temporal-difference* error. When our estimate of v_π is not correct, this method introduces a bias into the gradient estimation. If our estimate of v_π becomes more and more accurate, then this bias will asymptotically approach zero. We can accept this bias, as this method can further reduce the variance from the baseline. The reason this method is called *Actor-Critic*, is because we have our policy that takes actions in the environment (the actor), and a separate model to approximate v_π, which we call the *critic*. We usually have a separate loss for the critic which aims to minimise the following loss:

$$L(\phi) = \mathop{\mathbb{E}}_{\omega\sim\pi}\left[\frac{1}{2}\sum_{t=0} T(\omega) - 1(\gamma V_\phi(S_t) - v_\phi(S_t))^2\right], \tag{15.39}$$

where we approximate v_π using a function approximation V_ϕ with parameters ϕ.

We can see the effect of using actor critic in Figure 15.5, where the method caused a higher variance in comparison to the standard value-baseline method. However, we should note that we are calculating the gradient for an entire trajectory. Instead, Actor-Critic can be used to calculate gradients for single transitions, making it much more effective when training on continuing problems, whereas standard value-baseline, which uses Monte-Carlo returns cannot create an estimate from a single transition.

Actor-Critic is a type of bootstrapping technique, which can be used to increase the computational efficiency of the learning algorithms, but at the cost of introducing some bias. It is also important to

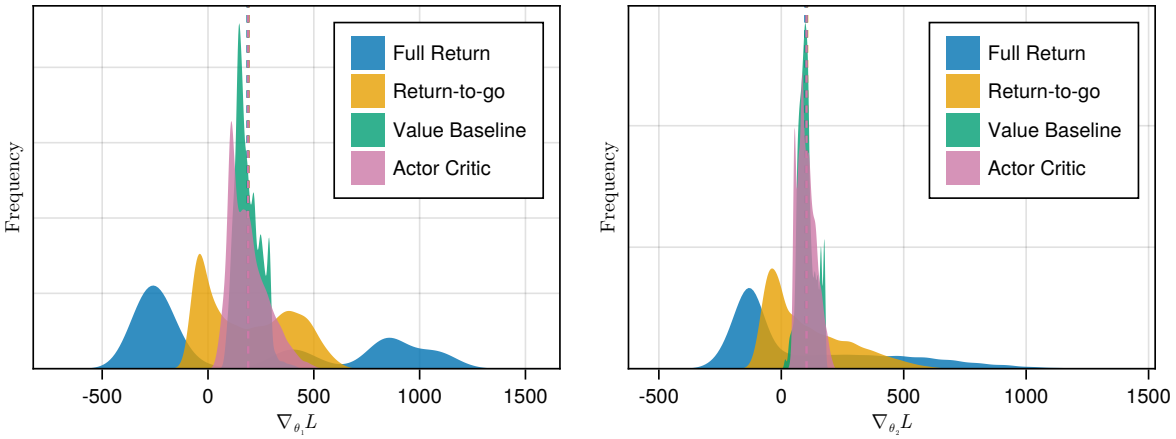

Figure 15.5: The probability density of the gradient estimates for θ_1 (left) and θ_2 (right), similar to Figure 15.4, with the addition of the Actor-Critic variance in pink. We can see that there is no bias when using an accurate value function, however, the variance for the trajectory is actually higher than just using a value baseline. However, this method allows one to sample single transitions from the trajectory, which can vastly reduce variance in online and continuing problems.

realise that the value function needs to be quite accurate for the bias to be reduced, so if the value function is too complex, then actor critic methods may not convergence depending on the function approximation used for the state. In contrast, value-baseline methods only affect the variance of the gradient estimate, not the mean, and therefore even if the value function is too complex to approximate exactly, no bias will be introduced. The only reason we did not see any introduced bias when using Actor-Critic in Figure 15.5, is because we used very accurate approximation for v_π as the critic.

15.6 Summary

In Part III, we have covered the basics of reinforcement learning and outlined a few methods which you can apply to real world probems. At the beginning, we outlined the overall task of trying to learn how to behave in a given situation such that we maximise some scalar return, which helps define our goals and tasks. From this point, we were able to rigorously reframe the problem in terms of predicting value functions and using these predictions to optimise our policy (how we took actions).

In the last chapter, we were able to depart from the standard approach of RL using value functions, and instead focus on optimising the policy directly via policy gradient methods. However, we found that the framework of value functions was still very useful, even when directly optimising the policy, as they allowed us to reduce the variance of our gradient estimate, and hence learn more efficiently.

In these notes, we have only scratched the surface of reinforcement learning, but have been able to learn the basics of the framework. Many of the nuances of the field are based in the techniques and the understanding gained over the past few chapters and form the bedrock of current cutting edge research in the field.

Bibliography

[1] M.L. Boas. *Mathematical Concepts in the Physical Sciences, 3rd Edition*. Wiley, 2005.

[2] A. Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610 (2022), pp. 47–53. URL: https://doi.org/10.1038/s41586-022-05172-4.

[3] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[4] H. van Hasselt, D. Borsa, and M. Hessel. *Reinforcement Learning Lecture Series*. 2021. URL: https://www.deepmind.com/learning-resources/reinforcement-learning-lecture-series-2021.

[5] D.P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980* (2015). URL: https://arxiv.org/abs/1412.6980.

[6] K.P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: probml.ai.

[7] A.Y. Ng and S. Russell. "Algorithms for inverse reinforcement learning". In: *Icml*. Vol. 1. 2000, p. 2. URL: https://ai.stanford.edu/~ang/papers/icml00-irl.pdf.

[8] J. Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588 (2020), pp. 604–609. URL: https://doi.org/10.1038/s41586-020-03051-4.

[9] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. URL: https://doi.org/10.1038/nature16961.

[10] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 1998. URL: https://books.google.co.uk/books?id=U57uDwAAQBAJ.