# Improving Duckworth-Lewis: Statistical Methods for Revising Score Targets in Limited-Overs Cricket

*Matthew Knowles*


Department of Mathematics,
University of York,
United Kingdom

*A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Mathematics*

## Abstract

Write this last

## Notes

All code written in support of this project can be found on GitHub at:
*https://github.com/mattknowles314/mastersThesis.*

# Contents

# List of Figures

# Chapter 1

# Background

## 1.1 Limited-Overs Cricket and the need for Duckworth-Lewis

Cricket is a game played by two teams, each with 11 players. The objective for the batting team is to score as many "runs" as possible without losing 10 of their batsmen[1], who can be "out" in a variety of ways. Cricket is played in "overs", each over lasting 6 deliveries. Traditionally, the game lasted for 4 days and there was no limit to the number of overs the bowling team could bowl.

In 1963, the cricketing calender in the UK had for the first time a different format of the game ammended to it. The "Gillette Cup" introduced a form of cricket wherein each team has 1 innings, lasting 50 overs. The idea was that this competition, coming to a conclusion within the space of a day, would increase spectator numbers and by extension, ticket sales.

However, given the tournament-based nature of this new competition, we have the natural need for a definitive result. This is something that is not always guarenteed in "first class" cricket, where draws are common. [2]. As such, in order to allow for a result to be determined when a game is cut short, the idea of target-revision was introduced. This meant that a team could be set a roughly equivalent run target off of a reducded number of overs that would still classify them as the winners.

## 1.2 Problems raised with DLS

In [**?**], the authors found that DLS has a bias towards not only the team batting first, but whoever won the coin toss at the start of the game. The winner of the toss chooses whether their side will bat or bowl in the first innings. They go onto to propose a simple extension for reducing these biases, but we won't cover that in this project.

## 1.3 Project Aims

bar

## 1.4 Review of current Literature

We begin the look at literature on this topic with the paper published by F. Duckworth himself in 1998 [**?**]. In this paper, Duckworth proposes the "D/L" method based on 5 principles. Hwever, the issue with the sentence that appears after defining the function $Z(u, w)$. "Comercial confidentiality prevents the disclosure of the mathematical definitions of these functions". The claim is that these functions have been defined via experimentation. This however gives rise to the first issue: the first T20 game of cricket was not played until 2003. So clearly, D/L was not designed with T20 in mind, and so the functions

[1] There have to be two batters on the pitch, so if 10 wickets are lost, it leaves one batter stranded.

[2] Note that "draw" and "tie" are not interchangable terminology in cricketing terms.

derived from experementation and research will not be accurate for T20 cricket.

In 2004, a year after the first T20 matches were played, Duckworth and Lewis published another paper [**?**], in which they report that the table used for calculating the method can be employed by

# Chapter 2

# Data

## 2.1 Data Origin

The primary source of data for carrying out this project was downloaded from "cricsheet"[1] and stored locally on a private server. In total there are 2167 individual matches of data. Each in a JSON format. These cover matches ranging from the $3^{rd}$ of January, 2004. Up to the $20^{th}$ of July, 2021.

## 2.2 Attributes

Each JSON file contains a considerable amount of metadata surrounding the match in question. Along with ball-by-ball data for the entire match. We have access to attributes such as the date, where the match was played, the entire teamsheet for both teams, who the officials were, who won- and by what margin, who won the toss; and many others.

We also have the ball-by-ball data. So for every ball bowled, it gives who were the striking and non-striking batsmen, how many runs were scored and how. It also details if a wicket was taken that ball, and how.

## 2.3 Pre-Processing

Many Python scripts were written to get data in an appropriate form for analysis.

## 2.4 Problems

When it comes to machine learning, the more data the better is a general rule. Now this can sometimes lead to sub-problems, such as overfitting. But on the whole. it is much better to have as much data as possible. We will be training a neural network on 1435 data points. Stictly speaking, this isn't a lot of data, but there isn't much that can be done about this due to the cricketing calender only having a certain number of limited-overs matches each year.

[1] https://cricsheet.org/

# Chapter 3

# The DLS Method in Detail

We now look at the mathematics behind the D/L, and DLS methods. D/L being the original method, and DLS the method that Stern helped to revise. In the original paper, the authors state "Commercial confidentiality prevents the disclosure of the mathematical definitions of these functions." [**?**]. Which is naturally a slight problem for this, but what we can do is instead use sample values based on data we have to look at how these functions behave, so all is certainly not lost.

## 3.1 Origins: Duckworth and Lewis

We begin by looking at the original paper. The first thing to establish is how many runs are scored, on average, in a given number of overs. This is given by the equation:

$$Z(u) = Z_0[1 - exp(-bu)] \tag{3.1}$$

Where u is the number of overs, b is the exponential decay constant, and $Z_0$ is the average total score in first class cricket, but with one-day rules imposed.

Now because we don't have access to actual values for $Z_0$ or b, a plot to see what equation **??** looks like was created by using 3 sample values for $Z_0$. For b, it was a process of trial and error to find a value that resulted in the graph having a similar shape to original figures in the D/L paper.

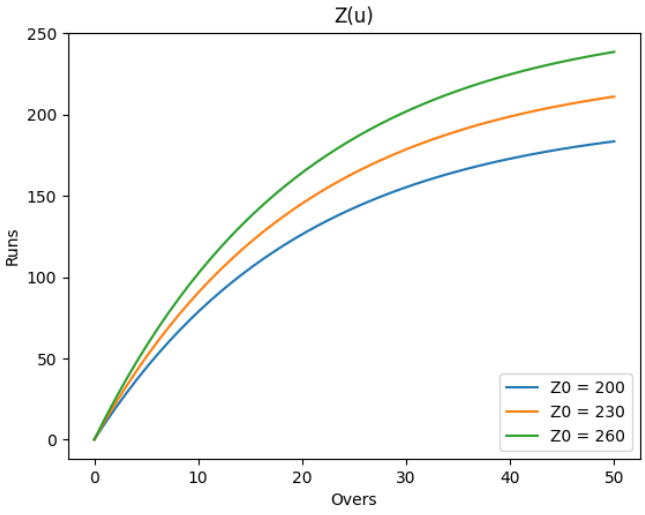


Figure 3.1: Graph showing how the rate at which runs are scored decays as a game progresses.

However, we have not yet looked at what happens when wickets are lost. To introduce this metric, equation **??** is revised to incorperate the scenario that w wickets have been lost, and that there are u overs remaining The revised equation is given as follows:

$$Z(u, w) = Z_0(w)[1 - exp(-b(w)u)] \tag{3.2}$$

Where now, we have $Z_0(w)$ giving the average total score from the last $10 - w$ wickets in first class cricket. We now also have $b(w)$ as the exponential decay constant, which now changes depending on wickets lost.

With this in mind, we now look at the specific case of equation **??** with $u = N$ and $w = 0$, namely, the conditions at the start of an N-over innings. We have:

$$Z(N, 0) = Z_0[1 - exp(-bN)]. \tag{3.3}$$

Which we then incorperate into the ratio

$$P(u, w) = \frac{Z(u, w)}{Z(N, 0)}. \tag{3.4}$$

The ratio **??** gives, keeping in mind there are u overs still to be bowled, with w wickets lost, the average proportion of the runs that still need to be scored in the innings. It is this ratio that is where the revised scores come from. Let us now look a bit more at how that works practically.

**Example 3.1.1.** *Assume there is a break in the second innings (due to rain or similar), which results in the second team missing some overs. Let $u_1, u_2$ be the number of overs played before the break, and available after it respectively. We impose the condition that $u_2 < u_1$. At the time of the break, the second team had lost w wickets. The aim is to adjst the required score to account for the $u_1 - u_2$ overs they have lost. The winning "resources" available are given by*

$$R_2 = [1 - P(u_1, w) + P(u_2, w)].$$

*Which means, if the first team batting scored S runs, then the new tartget is given by*

$$T = \lceil SR_2 \rceil$$

## 3.2 Improvements by Stern

We now look at the DLS method, introduced by Stern in [**?**] to extend the original D/L method. The motivation for DLS was that very high scoring cricket matches presented a pattern of straightening towards average run rate which is not uniform across the innings considered (around 500). To account for this, Stern introduces an additional damping factor to the equation for D/L. The updated equation is now given by:

$$Z_{dls}(u, w, \lambda) = Z_0 F_w \lambda^{n_w+1} \left\{1 - e^{-ubg(u,\lambda)/F_w \lambda_w^n}\right\}. \tag{3.5}$$

This equation assumes u overs remaining, w wickets down in an M-over match with a final total of $S$. Here, we have:

$$g(u, \lambda) = \left(\frac{u}{50}\right)^{-(1+\alpha_\lambda+\beta_\lambda u)} \tag{3.6}$$

In the above, we define $a_\lambda = -1/\{1+c_1(\lambda-1)e^{-c_2(\lambda-1)}\}$ and $\beta_\lambda = -c_3(\lambda-1)e^{-c_4(\lambda-1)}$. The constants $c_1, \ldots, c_4$ are based on what Stern describes as a "detailed examination" of high scoring cricket matches.

## 3.3 Extension: Bayesian Modelling

This section will look at the work of paper [**?**], which used Bayesian modelling to try and improve the score predictions of the D/L method. The authors used the same dataset as we are using for the dissertation, although over a slightly smaller range of games. Only games between 2005-2017 are used, whereas our dataset goes up to 2021. Note that to keep consistent with the fact we are talking about a different model now, we swich from using $Z(u, w)$, to using $R(u, w)$, as to keep consistency with the different papers being looked at. They start by introducing the following nonlinear regression model:

$$\bar{R}(u,w) \sim N(m(u,w;\theta), \frac{\sigma^2}{n_{uw}}) \tag{3.7}$$

Where $\bar{R}(u,w)$ is the sample average of runs scored by a team from the total number of matches in the data set. We have $m(u,w;\theta)$ as the corresponding modeled population average of runs scored by a team when a considerable amount of games are taken into account. $\theta$ denotes a vector of parameters that will be specified later on. Since $R(u,w)$ is not calculated in all games, the average score is taken over all matches where scores are present, this is given by $n_{uw}$. The sample mean $\bar{R}(u,w)$ is the calculated over this quantity. This is actually the point which motivates using Bayesian statistics to extend the D/L method. The authors report that approximately 26.8% of values for $R(u,w)$ were missing in the dataset, so by using a Bayesian inference model, we can use the posterior predictive distribution to account for missing values. Which in turn should increase the predictive accuracy of this model.

We return now to look at the $m(u,w;\theta)$ parameter that appears in **??**. This mean function, based on the plots produced in the original D/L paper, (see **??** for an example), the authors adopted an exponential decay. We can see why this choice makes sense from figure **??**. The resources considered by this method, namely runs and wickets, do decrease exponentially as a game progresses. $m(u,w;\theta)$ depends on the parameters $a_w = Z_0(w)$ and $b_w = b(w)$, each one depending on the number of wickets fallen at that given time, with u overs remaining. We therefore have:

$$m(u,w;\theta) = a_w(1 - exp(-b_w u)) \tag{3.8}$$

$$\theta = \{(a_w, b_w); w \in \mathcal{W} = \{0,1,\ldots,9\}\} \tag{3.9}$$

Before proceeding with defining the prior specifications on the parameters for this model, we need the following distribution:

**Definition 3.3.1.** *$X \sim Ga(a,b)$ has a **Gamma Distribution** with mean $\frac{a}{b}$ and variance $\frac{a}{b^2}$ if its probability density function is*

$$\frac{b^a}{\Gamma(a)} x^{a-1} e^{-bx}.$$

We can now define the prior specifications on the parameters. First, we fix $A_0$ and $B_0$ large enough such that $R(50,0) < A_0$. We initialise the model with $a_0 \sim U(0, A_0)$ and $b_0 \sim U(0, B_0)$. Then given any pair $(a_0, b_0)$ and for $w = 0, 1, \ldots, 8$, we generate:

$$a_{w+1}|\sigma^2, a_w, b_w \sim U(0, a_w) \tag{3.10}$$

$$b_{w+1}|\sigma^2, a_{w+1}, b_w, a_w \sim U\left(0, \frac{a_w b_w}{a_{w+1}}\right) \tag{3.11}$$

$$\frac{1}{\sigma^2} \sim Ga(a,b). \tag{3.12}$$

Above, U(a,b) represents a uniform distribution over the open interval $(a,b)$. Now that the prior distribution is obtained, the likelehood function is then given by:

$$L(\theta,\sigma^2) = \left(\frac{1}{\sigma/\sqrt{n_{uw}}}\right)^{500} exp\left\{-\frac{1}{2(\sigma^2/n_{uw})}\sum_{u=1}^{50}\sum_{w=0}^{9}(R(u,w) - m(u,a_w,b_w))^2\right\} \tag{3.13}$$

The authors chose their parameters $A_0$, $B_0$, $a$ and $b$ such that the prior is not sensitive to the posterior inderence. The values chosen were therefore $A_0 = 200$ $B_0 = 100$ and $a = b = 0.1$.

One thing briefly mentioned in chapter 3 of this paper is the average score at the fall of each wicket. This isn't discussed further, but it's interesting to see how it behaves. So we take a brief detour to look at it. This can be seen in figure **??**.

Initially, one may think that this is normally distributed. It certainly holds a bell curve-like shape. However, a Kolmogorov-Smirnov test [**?**] was performed to see if these values were normally distributed,

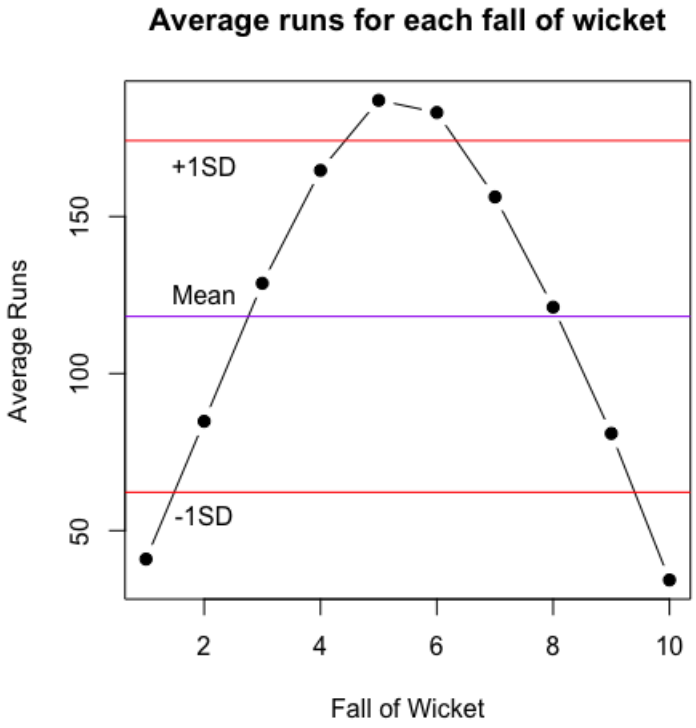


Figure 3.2: Plot of average runs scored at the fall of each wicket. Averages taken over 1437 innings.

and it turns out they aren't. This test was performed using the R function *ks.test()*, which returned a p-value of $2.2\text{x}10^{-16}$ for the two-sided alternative hypothesis parameter.

Does the shape of this curve make sense? Well yes, because usually batters 3/4 are actually the best in the side, so the fact they put on more runs is not a suprise. But what this does do is reinforce the idea behind using an exponential decay for modelling resources. This is because cumulatively, the tail end of batters (the last 4) will not put on as many runs as the higher/middle order.

Most of the work from this paper went into creating a better resource table for calculating revised scores. Which isn't particularly relevant to this project, but what is is the section on score prediction. The authors split every match at the $35^{th}$ over to try and predict the final score. They found the bayesian model gives a better prediction of final match scores. Figure 3 of their paper outlines these results.

# Chapter 4

# Exploratory Data Analysis

## 4.1 Fall of Wicket Densities

In this chapter, we are looking only at the first innings of the games, and only those games in which the full 50 overs were played. The reason for this is the models we will build are going to try and predict a score as if a full innings has been played.

We begin our exploration of the data with a look at how the density of the runs scored per fall of wicket changes. This has been done for each individual team in the dataset, and in figures **??-??**, we can see how this evolves.


Figure 4.1: Density of all teams for first wicket falling

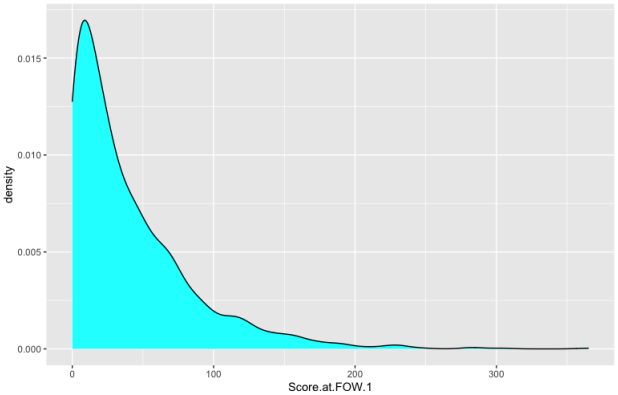


Figure 4.2: Overall density plot for FOW 1

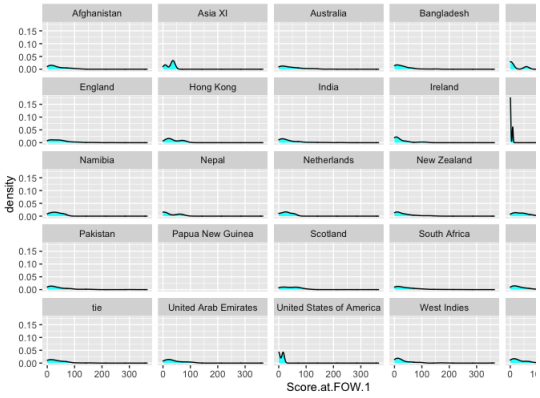


Figure 4.3: Density of all teams for fith wicket falling

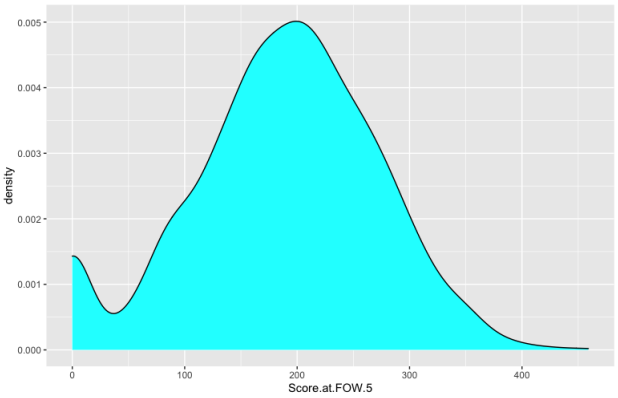


Figure 4.4: Overall density plot for FOW 5

In **??**, we see the density is heavily skewed to the left. This makes sense, as the bowling team will

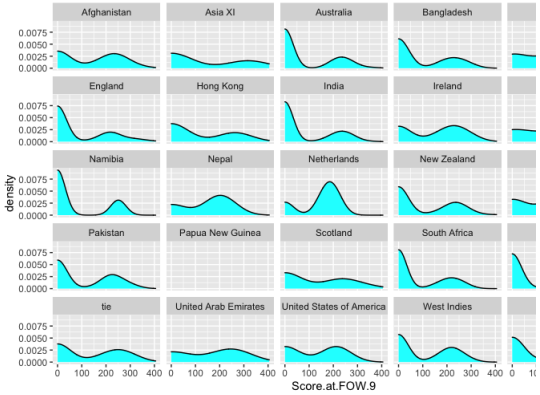


Figure 4.5: Density of all teams for ninth wicket falling

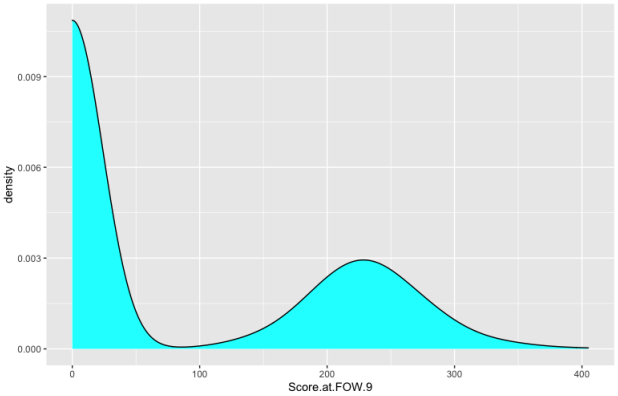


Figure 4.6: Overall density plot for FOW 9

presumably be starting their innings by using their best bowlers, who will be hunting to get wickets early on. In **??**, we see a much more normally distributed density function. But in actual fact, we see this interesting second, smaller peak appearing lower down in the score. Does this make sense? It's certainly not suprising. What these two peaks exemplify is the fact games can go heavily in favour of the bowling team, which can be seen in the first small peak, wherein they have taken a lot of wickets in quick succession, meaning the later order batters are coming in earlier than usual. Secondly, it shows when the batting team is having a good day, because we have this much larger peak around the 200 runs mark.

Finally, in **??**, we can see that the earlier bowling advantage peak is much higer, because the lower order batters are traditionally less skilled at batting, and so the bowling team have a distinct advantage in taking wickets against these players. But we also see the second, batting-favoured peak is no much lower. This corresponds to the scenario in which the earlier batters have laid a good foundation of the game, and the lower-order batters have not had to contribute much to the score.

## 4.2 Outliers in Runs Data

In the next chapter, it will be necessary to choose a loss function for improving the neural network that we create. To aid in determining this, we need to look at the spread of runs scored in a full innings of data. This can be seen in **??**.

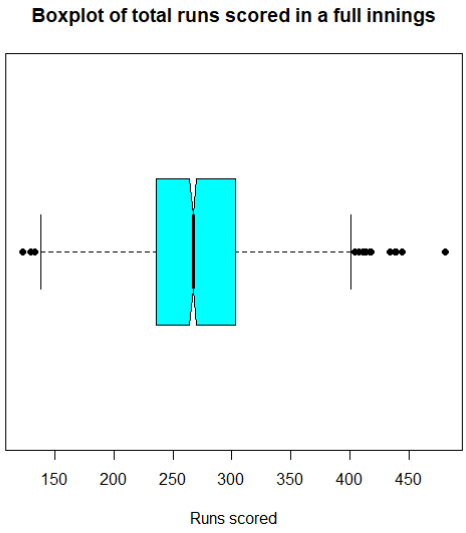


Figure 4.7: Boxplot showing the spread of runs scored

There are 363 data points greater than the third quartile, while 671 are below the first quartile. So 25.3% of our data lies above the third quartile, and 46.7% below the first. For that reason, we make the decision to use the "Huber Loss" function when building our neural network, since it is less sensitive to outliers.

# Chapter 5

# Overview of Pattern Recognition Techniques

In this chapter, we will discuss the mathematical foundations of the patern recognition methods that will be used to predict scores. There are two approaches to this. The first is to use already built packages for both a given programming language and simply feed data into it and present the results. The second method is to build them from scratch. The disadvantage to doing this is naturally that it is more time consuming. However, it does allow for more control over how our models work, and will allow us to understand our results better, than if we had used a proprietary model. This isn't to say that the code written for these models won't utilise packages for doing things like linear algebra calculations, but we wont simply import a neural network package and have results within 5 lines of code.

## 5.1 Neural Networks

### 5.1.1 A Brief Introduction to Neural Networks

Neural networks (NNs) have been the subject to a lot of hype in recent years. They are a machine learning method that is being applied to many problems in all sorts of fields. The network will be trained on runrate data, so for each game we have calculated the evolution of the runrate, and then we have the overall score for that game in the final column of the matrix. An example of a network can be seen in **??**. The first layer is the input layer, and the last layer gives the predictions. The middle layers are hidden and where the work of the network is done.

Let's look at what a NN actually looks like. The below is an example of a network with just one hidden layer.

We see there are $p$ input nodes at the bottom, M nodes in the hidden layer, and K output layers. The lines between the layers are given a "weight" and a "bias". These properties will be discussed in more detail shortly. The number of hidden layers to be used will be the subject of experimentation. In order to find try this out, we will have to randomly split the matrix into a training matrix and a testing matrix. We can then evaluate the network by trying to predict values from the testing set and seeing how well it does. This will then lead to us refining the number of hidden layers as appropriate.

One area of the world where neural networks are being applied to value prediction is in the stock market. Naturally if one can predict how the value of a stock will change over some time period, then one can protect themself from a bad investment, or profit heavily from a good one. We will use similar methods here for building our neural network. In [**?**], the authors twest two different Neural Netowrks, and find that using a "Multi-Layer Feed Forward Nerual Network" is the better choice for predicting how stock values will change. With these motivations in mind, we can begin to construct the networks.

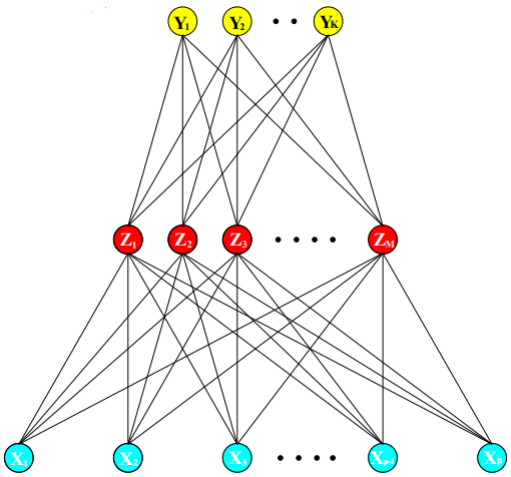

Figure 5.1: Example of a single hidden layer neural network, as in [**?**]

### 5.1.2 Builiding The Network

Our input layer will have 50 nodes, one for the runrate at the end of each over. We will begin with 5 hidden layers, although this is subject to change. The output layer will of course only have one node, the value of which will predict the score of the game.

We begin by looking at a single node. The proper name for each node is "perceptron". Each perceptron takes in the values of a vector, and an extra "+1" intercept term. The perceptron then outputs a valye h given by **??**:

$$h_{W,b}(x) = f(\mathbf{W}^T x) = f(\sum_{i=1}^{K} W_i x_i + b). \tag{5.1}$$

Here, $K \in \mathbb{N}$ is the number of elements in the vector, and $f : \mathbb{R} \to \mathbb{R}$ is called the activation function. There is a bit of choice in which activation function to use. The two common choices are:

$$f(z) = \frac{1}{1 + exp(-z)} \tag{5.2}$$

$$f(z) = tanh(z) \tag{5.3}$$

The comparrison of these two functions can be seen in **??**.

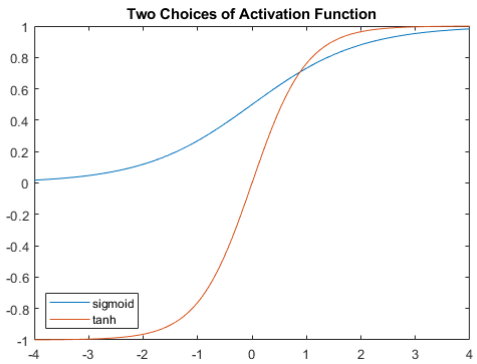


Figure 5.2: Graph showing the shape of different activation functions.

We will be using (4.3) as our activation but will return to the activation function shortly. For now, lets look at the network we're going to use. The initial neural network that we're building. The code for this activation can be seen in **??**.

```
def act(z):
    return math.tanh(z)
```

Figure 5.3: Python code for the activation function

We begin with getting from the input layer, denoted $L_0$. Each node in $L_0$ takes a value $a^{(0)i}$ for $i \in \{1, 2, \ldots, 50\}$. We use $textbf a^(0)$ to denote the vector containing these values. For every node in $L_0$, we have 25 connections coming away, one going to each of the nodes in $L_1$. 25 was an arbitraty choice for the size of $L_1$, and is subject to change based on results from initial testing. So that means we have $50 \times 25 = 1250$ weights for just the first layer alone. Some linear algebra is to be done here to get values for the vector $\mathbf{a}^{(1)}$. We use $w_ij$ to denote the weight from node j in one layer to node i in the prior layer. We can then construct a matrix containing the weights between $L_0$ and $L_1$, which we denote $W^{(1)}$. This matrix is given by **??**.

$$W^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & a_{1,50} \\ w_{2,1} & w_{2,2} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{25,1} & a_{25,2} & \cdots & a_{25,50} \end{bmatrix} \tag{5.4}$$

Using **??**, and combining with $a^{(0)}$, we can get the following:

$$A^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & a_{1,50} \\ w_{2,1} & w_{2,2} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{25,1} & a_{25,2} & \cdots & a_{25,50} \end{bmatrix} \times \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_{50}^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_{25}^{(0)} \end{bmatrix} \tag{5.5}$$

$$= W^{(1)}\mathbf{a}^0 + \mathbf{b}^1 \tag{5.6}$$

Where $\mathbf{b}^1$ is the vector containing the biases for each node. At this point, we are incredibly close to having values for $\mathbf{a}^{(1)}$. The last thing to do is apply the activation function. The above has resulted in a $25 \times 1$ vector, we use the notation $f(A^{(1)})$ to denote applying the activation function to each element in the vector $A^{(1)}$. Putting this all together, we have the following equation:

$$\mathbf{a}^{(1)} = f(W^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^1) \tag{5.7}$$

Which gives us values for all the nodes in the first hidden layer. This process is then repeated for all the the remaining layers in the network. Implementing this in Python is not too difficult, and the code can be seen in **??**. The only function that can be seen here with no code is *is_Valid()*, which just checks that all the constituents are of the right dimensions. If this returns false, then it wont let the calculations go ahead.

This process is then repeated for each layer, creating a diffferent weights and bias matrix going between each layer in the network. These matrices are naturally of different sizes, but the principle is exactly the same. When we start the training process for this network, we will only have random variables for the weights and biases. To gain better values that can predict accurate results in the future, we need to train the network by feeding it feature vectors and their associated outputs.

To allow the netowrk to be trained, we first must define a "cost function". The way we do this is to input a vector to the input layer, let the network produce a result, say $y'$, which initially will be hooribly wrong, and square the difference between this and the true value $y$ for that particluar training vector. Put more precisely, let $\mathbf{X}$ be a feature vector containing 50 elements, and $y$ the corresponding output. Then define the cost function, $C(X, y) := (y' - y)^2$. The lower the value of $C(X, y)$, the better the network has done at predicting. The average value of $C(X, y)$ for each X and y in our training set, is then a good measure of the network's performance.

```
def next_layer(W, a, b):
    '''
    Takes in a weight matrix W, the current layer of values, a, and the vector of
    biases. Outputs the vector of
    values corresponding to the next layer in the network.
    '''
    if is_Valid(W,len(a),len(b)):
        A = []
        for row in W:
            sum = 0
            for element in row:
                sum += element*a[row.index(element)]
            A.append(sum)
        for a in A:
            A[A.index(a)]= act(a + b[A.index(a)])
        return A
    else:
        return False
```

Figure 5.4: Python code for giving activation values to the next layer

The cost function is at the heart of how these networks "learn". All we're doing is minimising a cost function, to give matrices of weights and biases that produce the best output. The algorithm for minimising this cost function is called "gradient descent" [**?**]. Suppose we take every single weight and bias of our network, and turn it into a giant column vector.

**Example 5.1.1.** *In this first example, we initialise four random weights matrices, four bias vectors, and we run through the process outlined above. We know going into this that the cost is going to be high. Initialising the weights as random values* $w_{ij} \in [-1000, 1000]$*, we get an initial cost function for our full dataset of a huge 3098.4.*

We discussed in section **??** the need to use a Huber Loss function based on our data. Let us now define this function, as in [**?**].

**Definition 5.1.2.** *Given an estimation procedure* $f$*, the* ***Huber Loss Function*** *is given by:*

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta) & otherwise \end{cases}$$

Here, $a$ reffers to residuals, which is essentially our cost function. For that, reason, we can rewrite this loss function as

$$L_\delta(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & |y - y'| \leq \delta, \\ \delta(|y - y'| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{5.8}$$

Implementing this function is not difficult, we can do it in a few lines of Python code, as in **??**

```
def huber(y1,y,delta=1.0):
    if (np.abs(y-y1)) <= delta:
        return 0.5*(y-y1)**2
    else:
        return delta*(np.abs(y-y1)-0.5*delta)
```

Figure 5.5: Python code for giving activation values to the next layer

# Chapter 6

# Analysis and Results

## 6.1 Method 1

foo

# Chapter 7

# Conclusion

foo