Improving the effectiveness of Neural Networks in modelling defensive football teams in the Premier League

Harry Tabb

10635156

Mphys Project Report - Semester 2

School of Physics and Astronomy
The University of Manchester

May 2024

This Project was Performed in Collaboration with *Liam Keown* and *City Football Group*

Abstract

This paper explores methods to improve the predictions of defending player's positions in short phases of play in the Premier League using a time dependant Neural Network. These methods included cleaning the dataset, parameter and hyperparameter initialisation and different training techniques. This successfully replicated defensive positions to within an average of 2.6 m of the expected positions; however, failed to generalise to unseen sequences, predicting an average separation of 20 m.

Contents

1 Introduction

Artificial Intelligence has played an imperative role in recent developments across all industries. Pairing this with the increase in data available in the sports sector makes a powerful tool for analysis and prediction in the Premier League. This paper focuses on using machine learning to predict the positions of defending players based on the positions of the attacking players in short phases of play in football games.

In recent years, top Premier League teams have struggled to beat teams in the lower half of the table despite high possession statistics. This is attributed to the Park-the-Bus defensive style used by lower quality teams. This strategy involves focusing on the defensive structure by sitting deep and absorbing pressure against teams that can not be competed with on the ball. This tests the attacking team's ability to keep hold of the ball and work through a compact defensive structure which will likely end in a mistake allowing a counterattack. So, despite high possession statistics, top teams often drop points against teams that use this style of play. Therefore, simulating how highly defensive teams respond to different attacking styles is a powerful tool for finding weaknesses in this style of play.

In the previous report, player tracking data was used to train a time dependant Neural Network to predict defensive positions of [redacted] team between 2016 and 2020 [1]. The model successfully reproduced individual training sequences predicting positions to within an average of 1 m. However, it retained no memory between sequences in the training set and did not generalise to the testing data. The primary goal of this paper is to address the shortcomings of the previous model by solving three key issues: the small dataset size, the high variation in sequences within the dataset and the unsuitable training process. First, the dataset was expanded to include a wider range of teams. Then a new data filtering technique was developed to clean the dataset. Also, different training techniques were used to reduce the effects of overfitting.

2 Theory

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a subset of Neural Networks that are specialised to learn connections in time series data [2]. A classical Neural Network is composed of nodes grouped into three types of layers: the input layer, the output layer and the hidden layers. The activation of nodes in the input layer correspond to the input data, whilst the activation of nodes in the output layer correspond to the model's output. The nodes in the hidden layers represent patterns in the data. Each node is connected to all nodes in the subsequent layer by a weight and a bias. These are are responsible for manipulating the input to produce the output and are tuned during the training process. This is described fully in the previous report [1]. The key addition in an RNN is the feedback loop. This is an extra connection that allows the output at a given timestep, y_t, to be a function of the input at the current timestep, x_t, as well as the hidden output at the previous timestep, h_{t-1}. This is different from a normal Neural Network in which y_t is only a function of x_t. The hidden layer is an intermediate output that is used to calculate the current and next time step but does not directly correspond to the model's output. At a time step, t, the hidden layer is calculated using

$$h_t = \phi(x_t W_{xh} + h_{t-1} W_{hh} + b_h), \tag{1}$$

where ϕ is an arbitrary activation function introduced to add non-linearity to the model [3]. The model parameters are the input weight matrix, W_{xh}, the hidden weight matrix, W_{hh}, and the hidden bias vector, b_h, which are all tuned during the training process through backpropagation. The output at a given timestep is then calculated using

$$y_t = \psi(h_t W_{yh} + b_y), \tag{2}$$

where ψ is the activation function for the output, W_{yh} is the output weight matrix and b_y is the output bias vector. This process is illustrated in Figure 1.

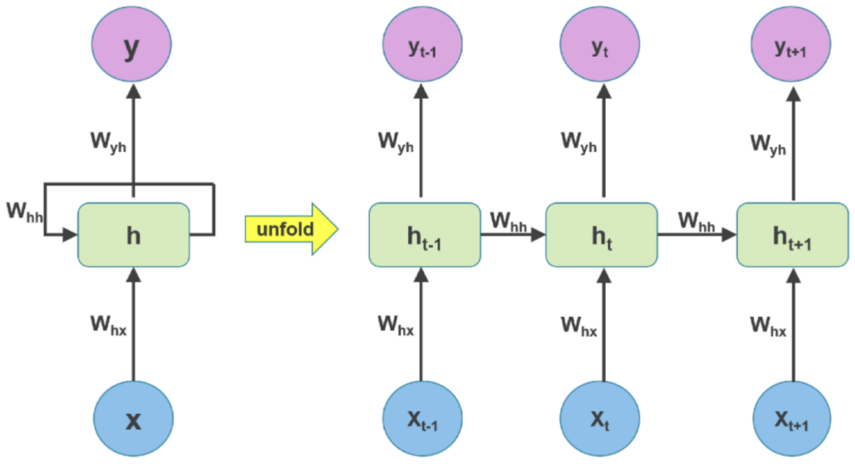


Figure 1: A schematic diagram showing the basic structure of an RNN. The left hand side shows the feedback loop going back into the cell and the right hand side shows this process unfolded [4].

RNNs struggle to learn connections between frames that have a large temporal separation [5]. This is caused by W_{hh}, which acts as a measure of how much the previous time step influences subsequent time steps, having to be constant between layers [6]. This causes issues during the backpropagation process to adjust the model parameters. The most common method for tuning parameters to minimise the loss is using gradient descent. For this process, the gradient of the loss function with respect to the model parameters, $\frac{dL}{d\theta}$, is used to calculate how to decrease the model parameters to minimise the loss. This is calculated using

$$\frac{dL}{d\theta} \; \alpha \sum_{1\leq j\leq t} (\frac{dL_t}{dh_t}\frac{dh_t}{dh_j}\frac{dh_j}{d\theta}), \tag{3}$$

where θ are the model parameters and

$$\frac{dh_t}{dh_j} \; \alpha \prod_{t\geq i\geq j} (W_{hh}\psi') \tag{4}$$

which can be seen from Equation 1 where ψ' is the derivative of the activation function [7]. Therefore, it can be seen from these equations that the gradient of the loss function is proportional to W_{hh}^T where T is the number of frames in the sequence. So, if $W_{hh} \geq 1$ or $W_{hh} \leq 1$, the gradient of the loss function will be very large or very small respectively. This means that the changes in parameter values will also be very large or small which will cause the minima of the cost function to be skipped over or never reached. This is known as the exploding and vanishing gradients problem which causes RNNs to be limited to roughly 5 to 10 time steps. The most popular method to overcome this is to use a Long Short Term Memory Network (LSTM) [8].

2.2 Long Short Term Memory Networks

An LSTM is a specific type of RNN that is better at learning long term dependencies as it overcomes the vanishing and exploding gradients problem. Therefore, this was the model architecture chosen for this project. The main difference between an LSTM and an RNN is the introduction of a more complex memory system that does not rely on a constant feedback weighting, W_{hh}. In an LSTM this is replaced by a memory cell, a schematic diagram of which is shown in Figure 2.

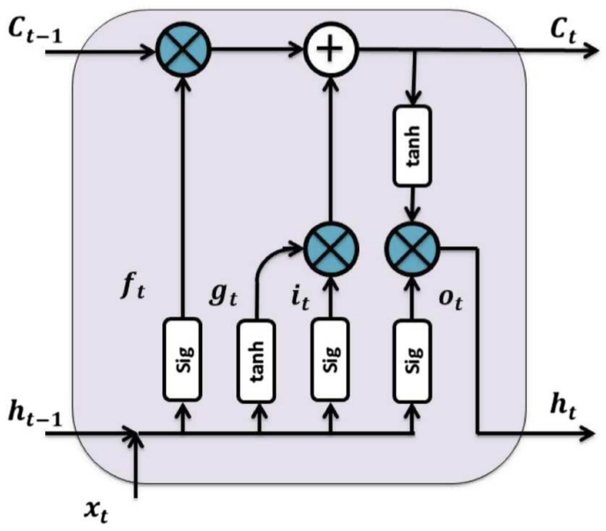


Figure 2: A schematic diagram showing the structure of a memory cell in an LSTM. Each gate is shown by the relevant activation functions [9].

The memory cell receives three input streams: the current input, x_t, the previous hidden output, h_{t-1}, and the long term memory, C_{t-1}. The current input and previous hidden output are passed through three gates that control the long term memory: the forget gate, f_t, the input gate, i_t, and the output gate, o_t. These are described by

$$P_t = \sigma(x_t W_{xP} + h_{t-1} W_{hP} + b_P), \tag{5}$$

where $P \in \{i, f, o\}$ describes each gate and W_{xP}, W_{hP} and b_P are the weight matrices and biases associated with the gate P [10]. Here, σ is the sigmoid activation function that compresses all outputs into the range (0, 1). The input node, g_t is defined similarly, but the sigmoid activation is replaced with a tanh function, T that returns values in the range (-1, 1). The sets of weights and biases for each gate are optimised during the backpropagation process analogously to the RNN process.

The forget gate controls how much of the long term memory is retained, whereas the input gate controls what parts of the current inputs are added to the long term memory. Therefore C_t is calculated by

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t, \tag{6}$$

where $\odot$ is the elementwise operator. The closer f_t is to one, the more of the long term memory will be retained. Whereas the closer i_t is to one, the more of the current input impacts the long term memory. This equation is key to overcoming the vanishing and exploding gradients problem as the model can change how much certain inputs impact the output. The long term memory also contains a feedback loop analogous to the RNN; however, in an LSTM, it includes a general activation function, $f_i(z(t))$, which is a function of the weighted input, $z_i(t)$, for a node i. To avoid vanishing or exploding gradients, the condition

$$f_i'(z(t))W_{ii} = 1, \tag{7}$$

has to be met, where f_i' is the derivative of the activation function and W_{ii} is the weight of the feedback loop. After integrating, this enforces that the activation function has to be linear and differentiable and the activation of the node i has to be constant in time. These are the key conditions for an LSTM to fix the vanishing and exploding gradients problem [11].

The output gate controls how much of the long term memory affects the current output. Mathematically, this is given by

$$h_t = o_t \odot T(C_t). \tag{8}$$

The closer o_t is to one, the more the long term memory impacts the current output.

3 Pre-Processing

3.1 Data Filtering

The raw coordinate data was extracted into two one-dimensional vectors of coordinates, as shown in Figure 3. The first vector, the model's input, contained the positions of the attacking players and the ball while the second, the model's output, contained the positions of the defending players. This process was repeated for every frame in the game, with a frame rate of 0.2 s.

$$\text{Input Vector: } I^{1\times 24} = [\ x_1,\ y_1,\ x_2,\ y_2,\ \ldots,\ x_{11},\ y_{11},\ x_{ball},\ y_{ball}\ ]$$
$$\text{Output Vector: } O^{1\times 22} = [\ x_1,\ y_1,\ x_2,\ y_2,\ \ldots,\ x_{11},\ y_{11}\ ]$$

Figure 3: An example of the input vector (first line) and output vector (second line) used in the model.

To begin the data processing, each game was first split up into phases of play. These were defined by periods in which one team was in constant possession of the ball. As only the defensive style of certain teams was of interest, only phases of play in which these teams were out of possession were kept in the dataset, all others were removed. After this, the alive conditions, summarised in Table 1, were applied. These were used to ensure that the game was underway during the phase of play due to some inconsistencies in the dataset. For example, sometimes the phase of play was not broken whilst the ball was out of the pitch bounds.

Alive Conditions
Ball within pitch bounds
22 Players within pitch bounds
Not a set piece
Not 10 frames after a set piece
Ball is not stationary for more than 10 seconds

Table 1: A table showing a list of the alive conditions used to remove unsuitable phases of play in the data processing.

The phases of play were then broken down into smaller 45 frame sequences which equated to 9 seconds of play. Frames within this sequence length were proven to be correlated in the previous report [1]. Any sequence longer than this included frames that were not correlated so would not have patterns that the model could learn. This was done by modelling the evolution of the game using a discrete time, homogenous Markov chain. This is defined as a stochastic process that evolves through a discrete set of states with time independent probabilities. A transition matrix, Q, was calculated to describe the evolution of the ball's position. This was then used to calculate how many frames it took the Markov process to reach a stationary state which was equivalent to the separation for frames to be uncorrelated.

The Park-the-Bus filter, described later in subsection 3.3, was then implemented to clean the dataset of sequences that were not defensive enough. This ensured that the sequences in the dataset were similar enough that the model could learn consistent relationships that were shared between all of the sequences. The coordinates were then normalised and standardised as described in Section 3.1 of the previous report, this transformed the pitch dimensions to match those of [redacted] so coordinates between sequences were comparable [12].

Finally, the attacking players were sorted into position groups (Goalkeeper, Defender, Midfielder, Attacker). Players within position groups were subsequently sorted by shirt number. This prevented players from moving around in the input sequence so the model could make connections between sequences based on indices in the input array. However, this had its limitations as it was not general between games or after player substitutions. Therefore, preferably, the attacking players would be sorted based on their exact positions such as "Left Centre Back"; however, due to limitations of the data provided, this was not possible. The whole data processing procedure is visualised in Figure 4 through a workflow diagram.

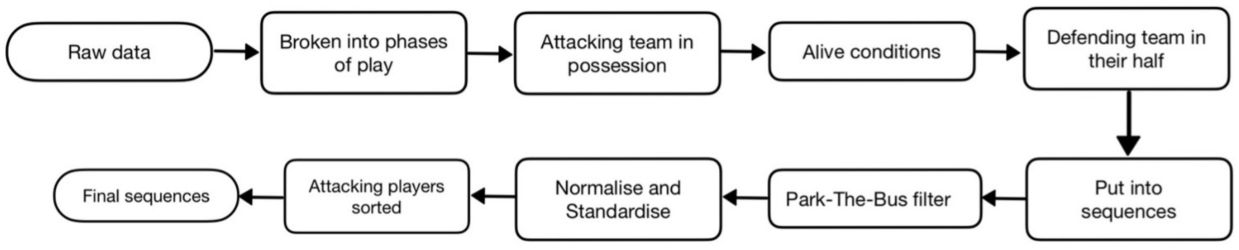

Raw data
Broken into phases of play
Attacking team in possession
Alive conditions
Defending team in their half
Put into sequences
Park-The-Bus filter
Normalise and Standardise
Attacking players sorted
Final sequences

Figure 4: A figure showing a workflow diagram of the data processing procedure.

3.2 Increasing the Size of the Dataset

As discussed previously, one of the reasons the model struggled to learn and retain memory between sequences was the lack of data the model was trained on. This is because football is an incredibly complex game, so it is expected that the long-term temporal relationship between the positions of the attacking and defending players is highly non-linear. Therefore, by giving the model a large dataset to train from, it would be expected that the model would start to learn patterns in the data instead of overfitting to individual sequences [13].

Previously, only defensive sequences from [redacted] were used. The [redacted] data was chosen due to their highly defensive style of play and they appeared in most seasons in the dataset so there was a large amount of data available. However, due to the cleaning process, there were only about 2,000 sequences in the final dataset, which was reduced to 1,600 using an 80/20 train-test split. This is widely considered to be the best ratio to reduce overfitting [14]. As the model was struggling to learn, the dataset was deemed to be too small. Therefore teams that had a similar style of play to [redacted] were also included to increase the size of the dataset.

Table 2 shows the teams that were included for each season and their average possessions across seasons. These teams were chosen as they all had an average possession below 45%, an arbitrary threshold set to define a defensive style of play. After including all of these teams, the dataset increased to 30,000 sequences.

Season	Team	Average Possession
2015/16	Leicester City	42.6%
	Sunderland	41.0%
	West Brom	39.8%
2016/17	Leicester City	41.9%
	Burnley	40.2%
	Sunderland	39.6%
	West Brom	37.2%
2017/18	West Ham	44.7%
	Burnley	43.5%
	Brighton	43.4%
	Newcastle	41.6%
	Stoke City	41.0%
	West Brom	40.3%
2018/19	Southampton	43.4%
	Brighton	42.0%
	Burnley	40.7%
	Newcastle	40.1%
	Cardiff City	34.9%
2019/20	Crystal Palace	44.5%
	West Ham	44.0%
	Bournemouth	43.9%
	Aston Villa	43.8%
	Sheffield United	42.9%
	Watford	42.5%
	Burnley	41.5%
	Newcastle	38.6%
2020/21	West Ham	42.8%
	Burnley	41.9%
	Sheffield United	41.4%
	Crystal Palace	40.2%
	Newcastle	38.3%
	West Brom	37.6%
2021/22	Brentford	44.6%
	Norwich	42.6%
	Watford	40.3%
	Newcastle	40.0%
	Everton	39.7%
	Burnley	39.7%

Table 2: A table showing the average possession across seasons for each team included in the dataset [15].

3.3 Park-the-Bus Filter

Increasing the size of the dataset can have negative effects. Despite the teams in Table 2 sharing similar possession statistics, they still play different styles of defensive football. This means that the sequences could be very different which makes it difficult for the model to learn patterns that are shared between sequences. In order for the model to learn well, the dataset should be large and sequences should be similar to avoid inaccurate analysis [16].

Previously, to define Park-the-Bus, only sequences in which all the defending team players remained in their own half for the whole sequence were included in the dataset. Although this was a reliable method to just include defensive phases of play, the sequences were still very different which led to a noisy dataset. For example, this method still included sequences that contained corners, free kicks and throw-ins where the defending team tended to cluster close together. These sequences were very hard to model as both the players and the ball tended to move in an erratic manner that was unique to individual sequences. Therefore, to help the model learn patterns, these had to be removed to reduce the dataset to only sequences that resembled a Park-the-Bus style in open play.

A Park-the-Bus style proved to be difficult to define quantitatively due to rotations and translations of the players across the pitch. Therefore a Park-the-Bus filter was made from games within the dataset. [redacted]

[redacted] This extremely low possession combined with [redacted] play style made this an ideal game to define a Park-the-Bus style from.

To do this, sequences were extracted from the game using the same selection cuts explained in subsection 3.1. Also, to remove the majority of deep set pieces, an extra selection cut was introduced. This removed any sequences where more than 6 defenders were in the penalty box at one time. This was an arbitrary choice but worked well to maintain a large dataset and remove most of the deep set pieces. From the remaining sequences, the mean positions of each defending player were calculated across the game. Then, a region was defined around each point that included 70% of data points from the game. The distributions of coordinates were highly non-Gaussian as shown in Figure 5. This can be seen by eye for this example; however, this was not the case for every player. Therefore the Kolmogorov-Smirnov (K-S) test was used to compare the distributions to a perfect Gaussian. This calculates the largest deviation from the cumulative normal distribution [18]. The calculated K-S statistic, D_{calc} is shown beneath each histogram and was compared to the critical value, D_{crit}. This was calculated to be $D_{crit} = 0.018$ using,

$$D_{crit} = \frac{1.358}{\sqrt{N}}, \tag{9}$$

for a significance level of $\alpha = 0.05$ where N is the number of points in the dataset [19]. If $D_{calc} < D_{crit}$ then the distribution is statistically Gaussian. This was not the case for any of the player position distributions so one standard deviation from the mean did not include 68% of the data points, following the Empirical Rule [20]. Therefore, to be conservative, Chebyshev's inequality was used to estimate the percentage of data within a standard deviation. This states that for a wide range of probability distributions,

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}, \tag{10}$$

where X is a random variable, μ is the mean, σ is the standard deviation and k is a positive constant [21]. Therefore, it can be calculated using Equation 10 that a minimum of 84% of data points are within 2.5σ of the mean for both the x and y distributions. The x and y positions of each player were also proven to be uncorrelated so the elliptical region was calculated to contain roughly 70% of data points by multiplying the percentage of data points in each one dimensional distribution. This confidence level was chosen as it closely resembles the 68%, 1σ, region for a Gaussian distribution.

These regions are shown for each player in Figure 6 by dashed ellipses which in turn are used to define a larger, Park-the-Bus, region. This was defined using the four defenders and four midfielders as the attackers' contribution to the defensive style varied too much between teams and seasons. On top of this, the difference between the average y position of the defenders and the midfielders from the [redacted] game was calculated to be $\Delta y = 6.30 \pm 9.20$ m. The errors for the average positions of the defence and midfield were equivalent to 1.75σ using Equation 10 to include 70% of the data. These were then combined in quadrature to find the error on Δy. For an individual sequence, if each player's mean position was within the Park-the-Bus region and if the average difference between the defence and midfield was within the error of Δy then the sequence was accepted as Park-the-Bus. This narrowed the dataset down to 16,000 sequences, but the quality of the data was improved which should improve the model's predictions.

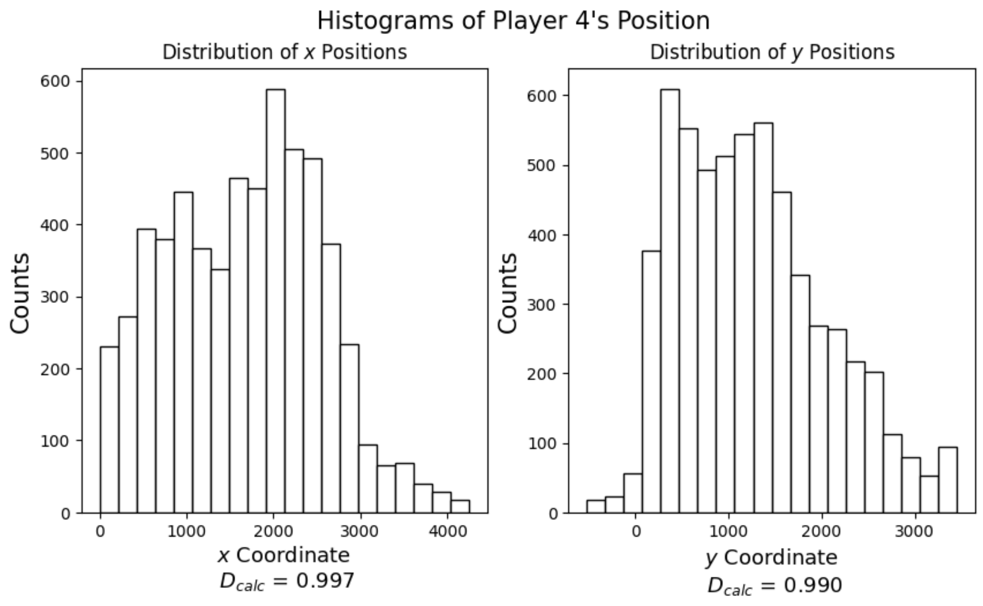


Figure 5: A figure showing histograms of a player's position in the x and y directions. The calculated D value from the Kolmogorov-Smirnov test is shown below each plot.

4 Results

4.1 Training Technique

The previous training process used a 10,000 epoch cycle. This meant that each sequence was seen 10,000 times before being shown the next sequence in the dataset. The results from this are shown in Figure 7a. Each sequence was learnt well, reaching a log loss, $\mathcal{L}$, between 8.0 and 10.0. The loss function is used as a quality of measure between the model output and the expected output. This was calculated using a Mean Squared Error (MSE) for each player coordinate averaged over all frames in a sequence. An angular cost was also used to influence predictions to be on a line connecting the centre of the goal and the expected position. This was to ensure the predicted position was still a good defending position if the MSE cost was not perfect. However, this was weighted so it only influenced the cost when predictions were within a metre of the expected position. This was described fully in the previous paper [1]. A good log loss was defined to be $\mathcal{L}_{thresh} = 10.0$ as this corresponded to an average player separation of approximately 2 m.

Despite reaching a good log loss, it spiked to between $\mathcal{L} = 13.0$ and $\mathcal{L} = 16.0$ between sequences. This showed that after learning one sequence, the model could not make a good prediction for the next sequence. This meant that the model was not finding patterns within the data and instead was just learning how to predict the desired output. On top of this, after the model was shown a new sequence it retained no memory of previous sequences. To fix these issues, a sequential training method was introduced. The model only saw a sequence for a singular epoch before being shown the next. The dataset was then cycled over so each sequence was seen the same amount as before. The results from this technique are shown in Figure 7b. More sequences were used in Figure 7b so the graph was compressed, but it is clear to see that between sequences the log loss oscillated much less ranging between $13.0 \leq \mathcal{L} \leq 14.0$. The lack of spikes in the loss is a clear sign that the model was no longer overfitting to a singular sequence. However, the minimum log loss was $\mathcal{L}_{min} = 13.0$ which was far above $\mathcal{L}_{thresh}$. This showed that the model was no longer learning sequences well. There was also no downward trend over the training process. The model should learn similar features between sequences

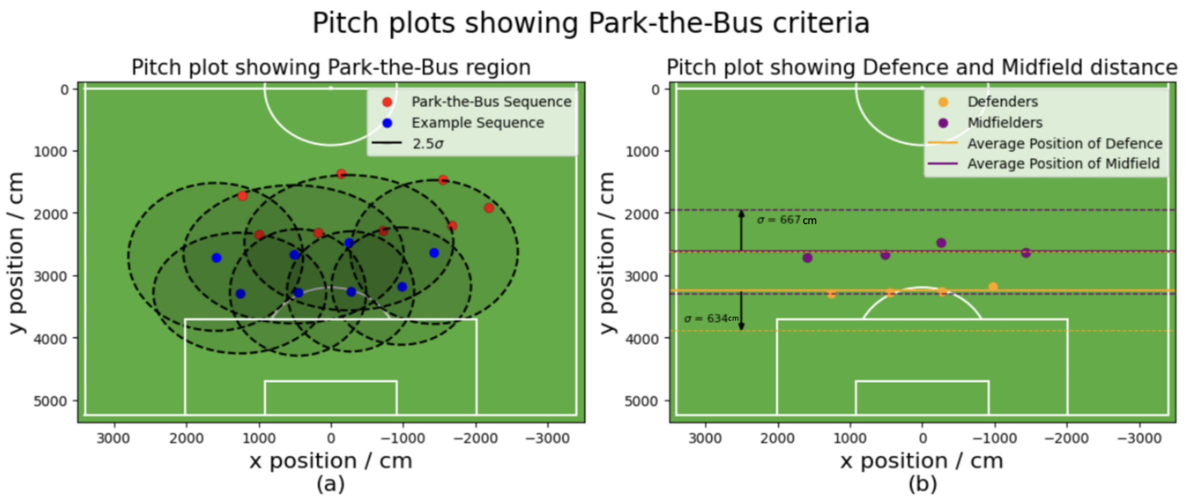
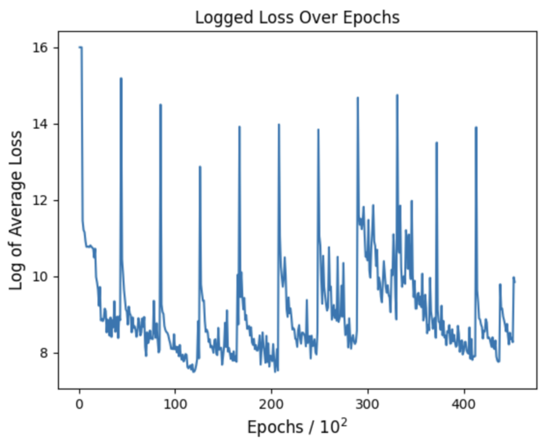
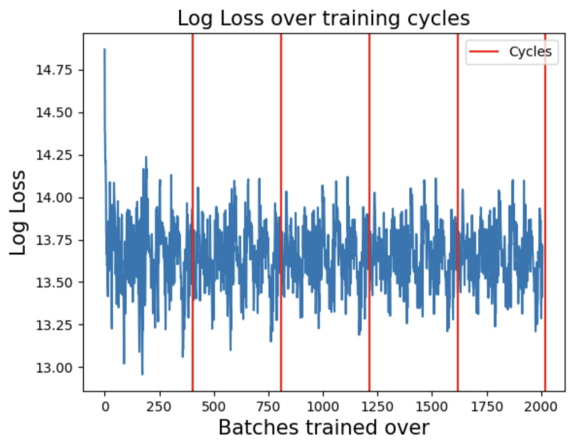


Figure 6: A figure showing the Park-the-Bus criteria used in the data processing procedure. Plot (a) shows the Park-the-Bus region with the mean positions of the players from this game plotted in blue and the error regions shown by the dashed black ellipses. An example sequence is shown in red. Plot (b) shows the average positions of the midfield and the defence along with the associated errors.

decreasing the loss; however, it still had no memory of previous sequences as similar features were not learnt.



(a) Log loss graph over epochs for 12 batches using the old training technique.



(b) Log loss graph over epochs for 400 batches using the new training technique. Each cycle is divided with a red vertical line.

Figure 7: Graphs showing the log loss for the old training technique (left) and the new training technique (right).

The model's outputs were plotted for a range of training sequences at the end of the training cycle. Two examples of these are shown in Figure 8. It can be seen that the model predicted a similar output regardless of the input. This prediction was a rough average of all the defensive sequences in the dataset which explained why the loss was not decreasing. By inspecting the parameters after training, it was found that this was caused by large biases in the final fully connected layer, b_y, compared to the size of the weights. This caused the output to be independent of the input as the second term in Equation 2 was much larger than the first term. Therefore, the biases, which are constant regardless of input, dominate the prediction with small variations caused by the weights.

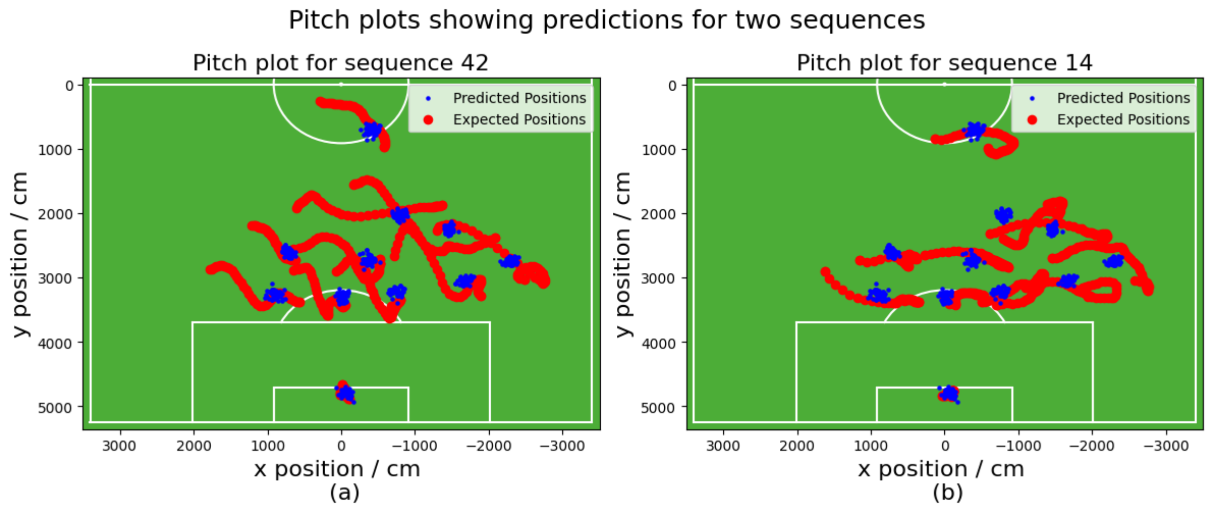


Figure 8: A figure showing the model's prediction for two sequences in the training set using the new training process.

4.2 Parameter Initialisation

To understand why the model was not learning, predictions from training on a singular sequence were plotted for different stages in the training process. These are shown in Figure 9. The model started the training process by predicting every point in the sequence at the origin. It then started learning by branching player paths out in straight lines from the origin. These were then narrowed down to the average of each path before branching out to learn the whole path. This was an inefficient way to learn as it took 20,000 epochs to reduce these lines to singular points; however, it then very quickly predicted the paths within 3,000 epochs. The initialisation of the model parameters was then investigated to speed up this learning process.

Previously, the model's weights were initialised by taking random numbers from a normal distribution with variance $\frac{2}{n_{in}}$, where n_{in} is the number of input neurons. This is known as He initialisation and was chosen as it works best with the ReLU activation function [22]. Zero initialisation was used for the biases; however, this was causing the model's initial prediction to be at the origin. Therefore, to improve convergence, the model was trained on a stationary sequence first. This meant that the start of each player path was already in a formation instead of being clustered together. This would hopefully influence the model to maintain a similar formation throughout the sequence. Also, this initialisation meant that the model already had some temporal dependence before training which will influence the model to keep the players moving between frames instead of staying at the same point as in Figure 8.

Pitch plots for training on a singular sequence with the new initialisation are shown in Figure 10. Instead of branching out from the origin in straight lines, the existing player paths were narrowed down to singular points. Therefore, the model lost all temporal dependence that was given in the initial parameters. However, it reached this point in 1,400 epochs which was much faster than the zero initialisation. However, the overall time to reach $\mathcal{L} = 10.0$ was roughly constant at around 22,000 epochs. As a good loss was reached faster and a low number of epochs are used in the training process, this initialisation technique was used over zero initialisation. However, this did not make much difference when applying this method to multiple sequences as the model still lost all knowledge of previous sequences.

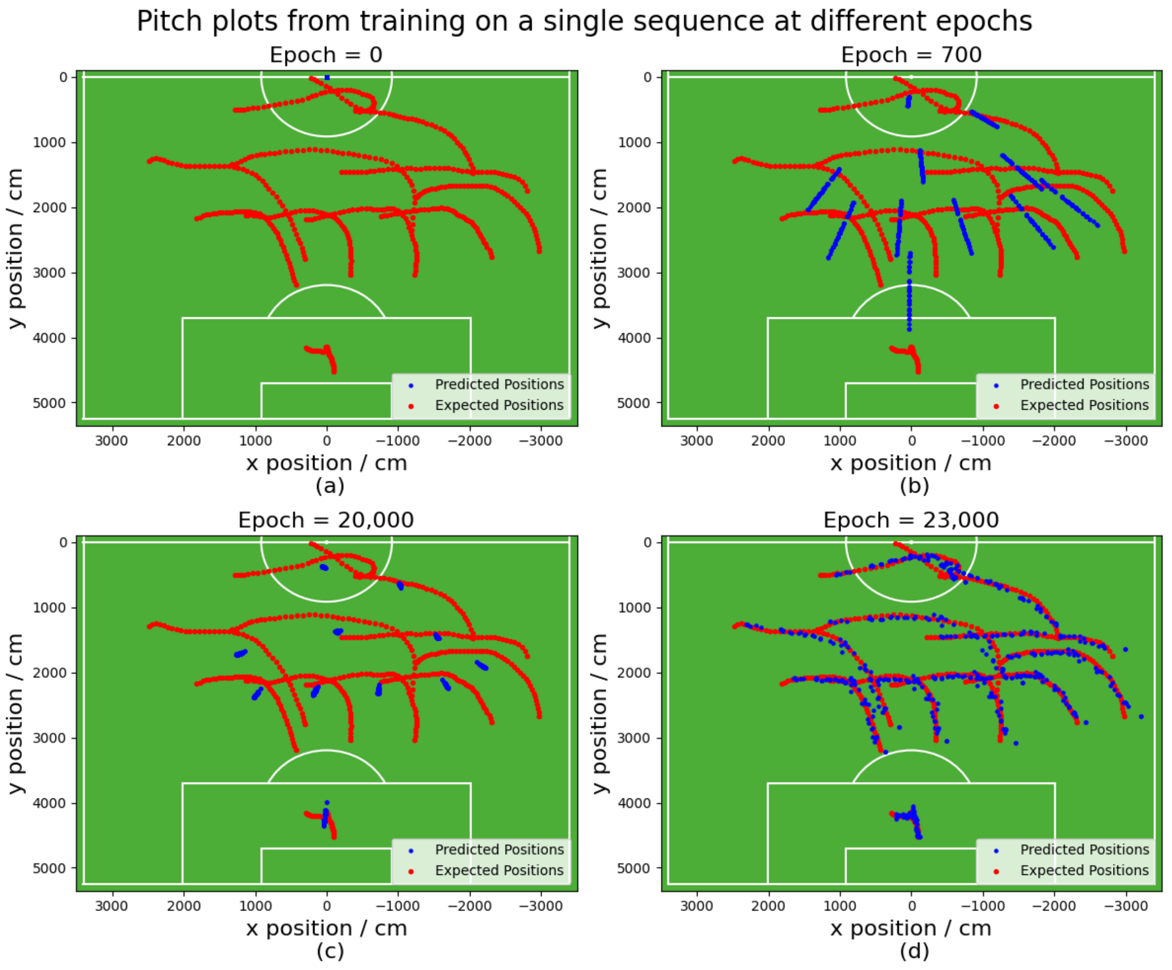


Figure 9: A figure showing different stages of the learning process after using He initialisation for the weights and zero initialisation for the biases.

4.3 Stationary Sequence

A potential issue was the problem being too complex for the model. Therefore, to simplify, a new model was made to train on individual frames. This removed any time dependence, so the problem was much simpler. This also meant that the model was reduced to a simple Neural Network instead of an LSTM. A singular frame was taken randomly from each sequence to remove the chance of any bias in the data. Also, the model was trained on every frame at the same time instead of one at a time. Figure 11a shows the loss graph for this process. The model learnt the sequences quickly, reaching $\mathcal{L}_{thresh}$ within 700 epochs. It also successfully made different predictions for different inputs as shown in Figure 12 which was an improvement from Figure 8. However, it was heavily overfitting to the training data shown by the test log loss plateauing at $\mathcal{L} = 14.0$. This meant that the model was too complicated and learnt to fit to noise in the data rather than finding meaningful relationships between frames. One way to reduce overfitting is to reduce the complexity of the model. However, the model still needs to be complex enough to learn the features in the data. Therefore, the number of layers in the model was reduced to reduce the complexity. Also, a dropout layer was added which allowed the model to selectively remove nodes that were causing overfitting. These methods are fully described in subsection 4.5. The loss graph after making these changes is shown in Figure 11b. The testing loss initially decreased with the training loss which was an improvement from Figure 11a. However, it then plateaued at $\mathcal{L} = 13.5$. Also, the training loss started to reach an asymptote above $\mathcal{L} = 13.0$. So despite making the model less complex to reduce overfitting,

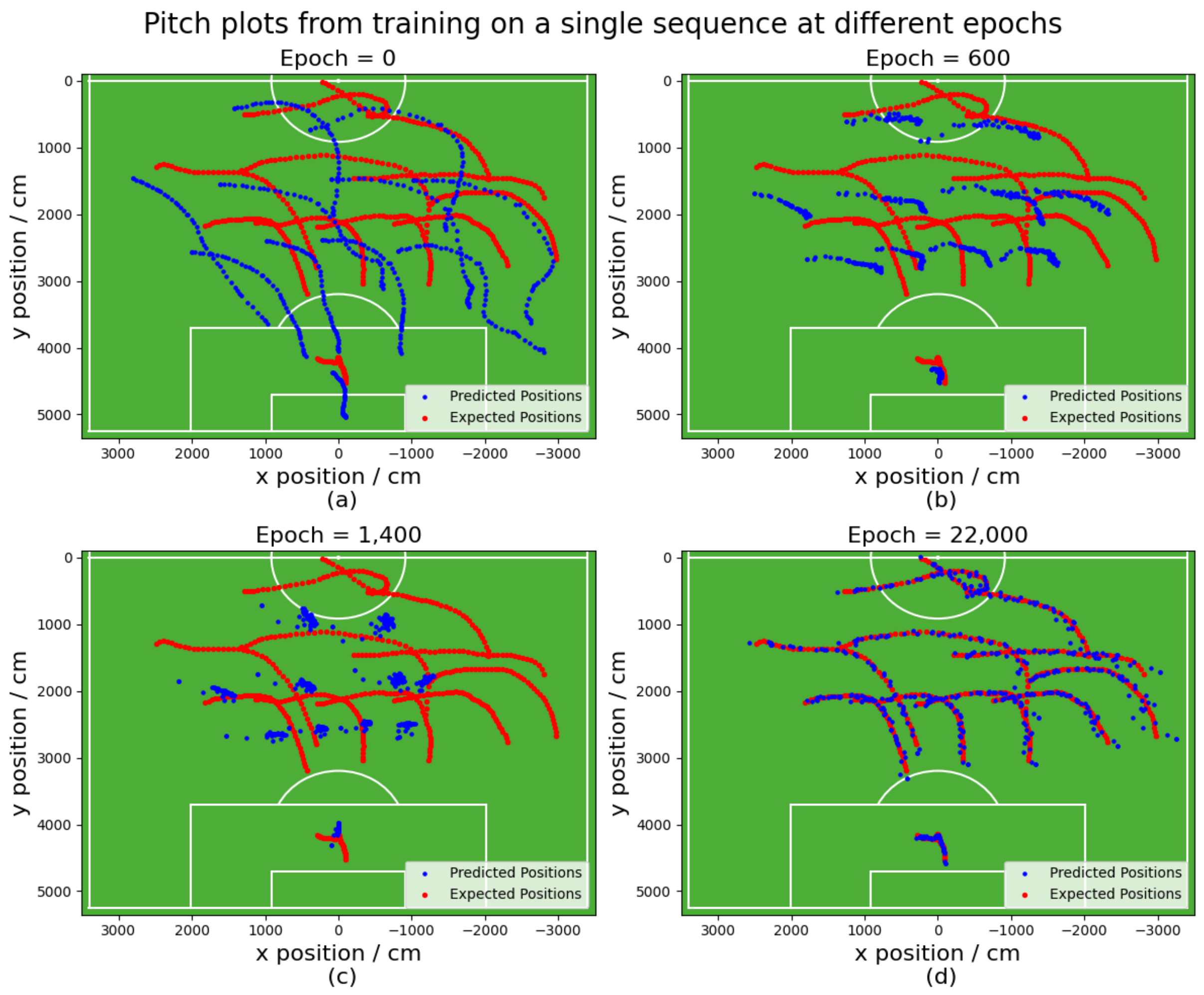
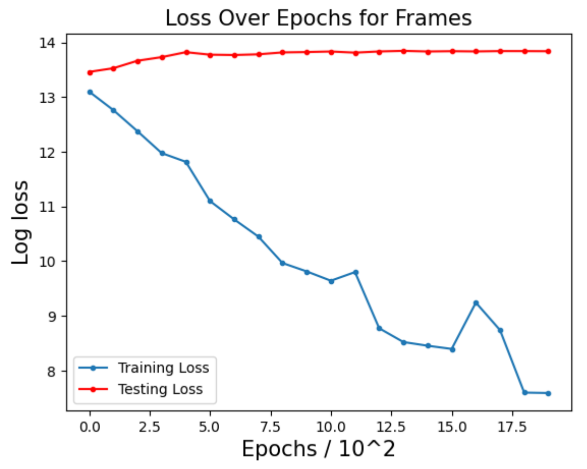


Figure 10: A figure showing different stages of the learning process after initialising the model with pretrained parameters.

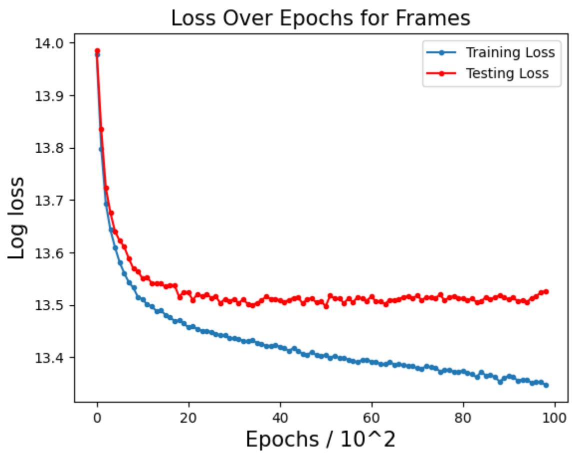
the model was not complex enough to reach a loss below $\mathcal{L}_{thresh}$. Therefore, a combination of hyperparameters to minimise overfitting but maintain good predictions needed to be found.

4.4 Reapplying to Sequences

Allowing the model to see every frame at the same time worked well for the stationary network, so this method was applied to the LSTM to see if it gave any improvements. A loss graph for this training process is shown in Figure 13a. After training over 20,000 epochs, the model predicted training sequences with an average log loss below $\mathcal{L} = 11.0$. This was not below $\mathcal{L}_{Thresh}$; however, this was an average over all sequences in the dataset, so some sequences had a loss below $\mathcal{L}_{Thresh}$ and some were above. The graph shows a distinct plateau at $\mathcal{L} = 13.8$, which was most likely a local minimum. The model then found its way out of this minimum after about 500 epochs shown by a sharp decrease in loss down to $\mathcal{L} = 13.3$. It remained at this loss for another 1,000 epochs before it kept learning until the end of the training cycle. At this point, the model looked like it was still learning but would soon reach an asymptote. Towards the end of the training cycle, the loss started to spike. This could be caused by the model trying to leave a local minimum with a high learning rate so it is jumping too far or the model having to first increase the loss to access a lower minimum. An example of a model prediction for a sequence in the training set is shown in Figure 13b. This sequence had an log loss of $\mathcal{L} = 10.5$ which corresponds to an average separation of 2.6 m over the sequence. The predicted paths



(a) Log loss graph demonstrating fitting well to the training set but badly to the testing set for training on frames.



(b) Log loss graph demonstrating improvements to the testing performance after tuning hyperparameters.

Figure 11: Graphs showing the log loss after training on individual frames. Both graphs show the model overfitting but at different points in the training cycle.

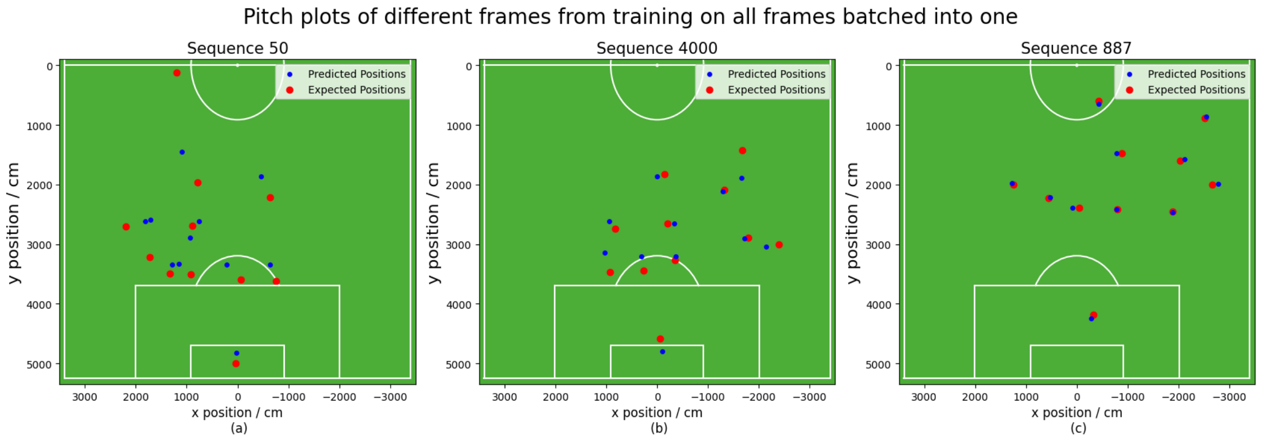
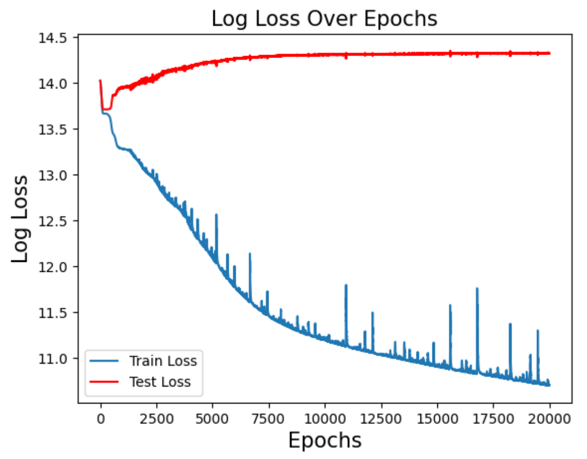


Figure 12: A diagram showing three pitch plots for different frames after training on frames instead of sequences.

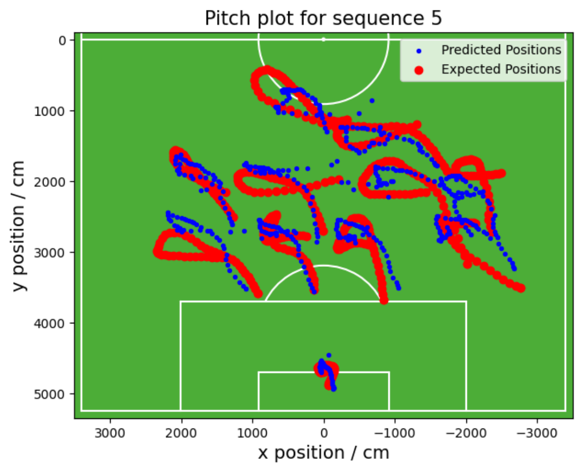
were roughly the same shape as the expectation which was an improvement from Figure 8; however, each path was slightly offset. This could be fixed if the model was trained for more epochs. However, despite the low training loss, the model was heavily overfitting to the training set shown by the constant test loss of $\mathcal{L} = 14.5$. For an unseen sequence, the model predicted a cloud of points roughly centered around the average position of the expected distribution as shown in Figure 14. Therefore, the model had a basic understanding of where the defending team as a whole should be based on the attacking players; however, it failed to predict any individual player paths. To improve the model's performance, solutions to overfitting had to be introduced.

4.5 Reducing Overfitting

Reducing overfitting is a big problem in machine learning. In general, overfitting occurs because the model is too complicated compared to the complexity of the patterns in the data [23]. Therefore, the model fits to noise in the dataset instead of the underlying relationships between sequences. This means that the model will replicate the training data very well, but will not



(a) Log loss graph demonstrating fitting well to the training set but badly to the testing set for training on sequences.



(b) Pitch plot showing the predicted and expected player paths for a chosen sequence. The log loss of this sequence was $\mathcal{L} = 10.5$.

Figure 13: Graphs showing the log loss after training on sequences (a) and a pitch plot showing the prediction for a sequence in the dataset (b).

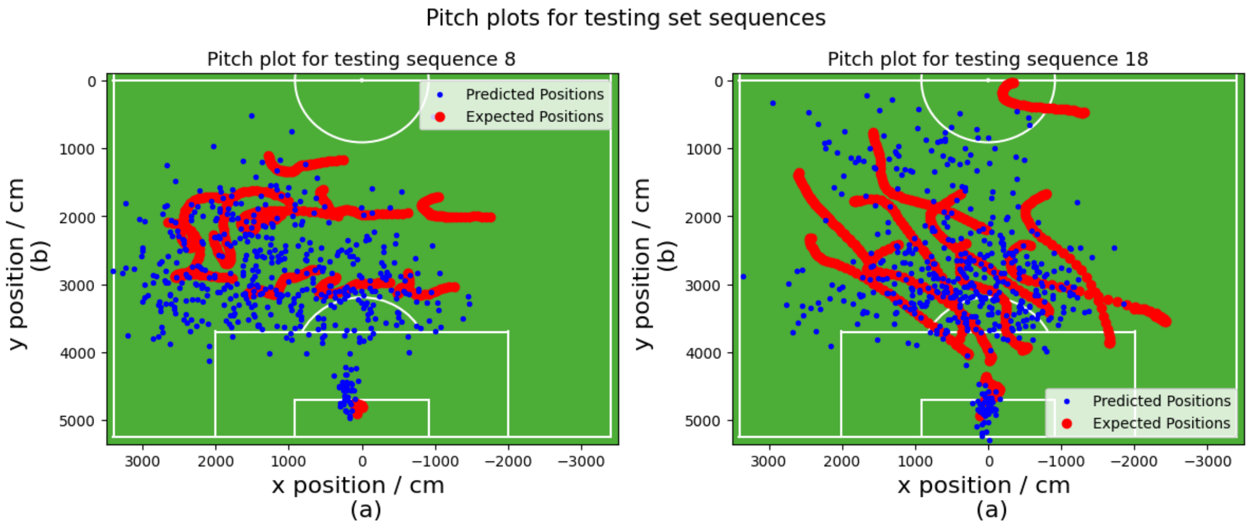


Figure 14: A diagram showing the model's prediction for testing sequences 8 (a) and 18 (b). Both sequences had a log loss of $\mathcal{L} = 14.4$.

make good predictions for the testing data. To stop this from happening, the complexity of the model has to be reduced; however, the model still needs to be complex enough to learn patterns in the data.

One method used to reduce the complexity was adding a dropout layer to the model. When the model is overly complex, nodes learn to fix mistakes caused by other nodes instead of fixing the errors causing the mistakes. This means nodes become codependent on each other which over complicates the network [24]. A dropout layer randomly selects nodes to be removed from the network which stops codependency and reduces the complexity. This is better than decreasing the complexity of the model before the training process as it allows the model to selectively remove nodes that are not needed to learn patterns. The dropout probability is another hyperparameter that needs to be tuned to reduce overfitting but maintain good predictions. However, implementing a dropout layer gave no improvements to the model. The test loss continued to plateau at $\mathcal{L}_{test} = 14.3$ but analogously to the stationary sequence, the training loss asymptoted above $\mathcal{L}_{test} = 13.0$. Therefore, it was concluded that the tuning of the hyperparameters would not fix the overfitting as a drop in complexity prevented the training set from being learnt. However, the final hyperparameters that were used are shown in the appendix.

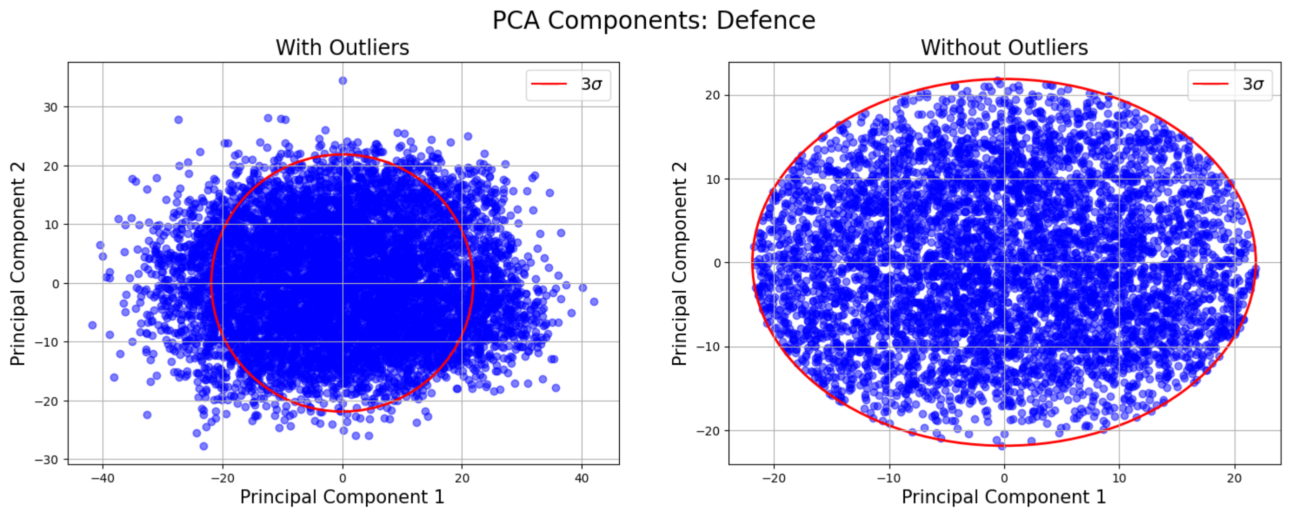
A potential problem that was causing the model to overfit to the training set was the sorting of the input vector. As spoken about in subsection 3.1, the order of the coordinates in the input array were sorted into position groups and then shirt number. This ensured that each player was in the same index within frames; however, this was not the case between sequences. The effect of this was minimised by removing any sequences after a substitution was made. However, the problem was still apparent between games due to squad rotation and different formations. Instead of learning the relationships between the attacking and defending players across sequences, the model might have been learning relationships between indices in the arrays. This would cause issues as the player in each index was changing between sequences. Evidence that shows this could be the case is shown in Figure 14. Despite the erratic predictions of the outfield players positions for unseen data, the prediction for the goalkeeper is much more accurate. This could be because the goalkeeper is the only player that is consistently in the same index in the input vector; however, it could also be due to small variations in the goalkeeper's position between sequences.

As the hyperparameters could not be tuned enough to reduce overfitting and maintain a good prediction, it was thought that the dataset may still be too complicated. Therefore, Principal Component Analysis (PCA) was introduced to add a further data processing stage to find outliers in the data. This is a technique to reduce multidimensional data to the two most important components, the principal components [25]. So using this, each attacking and defending sequence was reduced down to a point in two dimensional space. These plots are shown in Figure 15. The standard deviation for each distribution was calculated and a 3σ level was drawn on each figure, shown by the red circle. This level was chosen as outliers are defined as beyond 3σ from the mean [26]. Each point outside of this boundary was removed from both the attacking and defending datasets. This reduced the number of sequences to 10,000 but they were more similar. The model was then retrained using the tuned hyperparameters from before with this new dataset. However, once again, this gave no significant improvement to the model predictions. Both the training and testing loss remained at the same values as before. This showed that removing outliers from the dataset gave no significant improvement to the model.

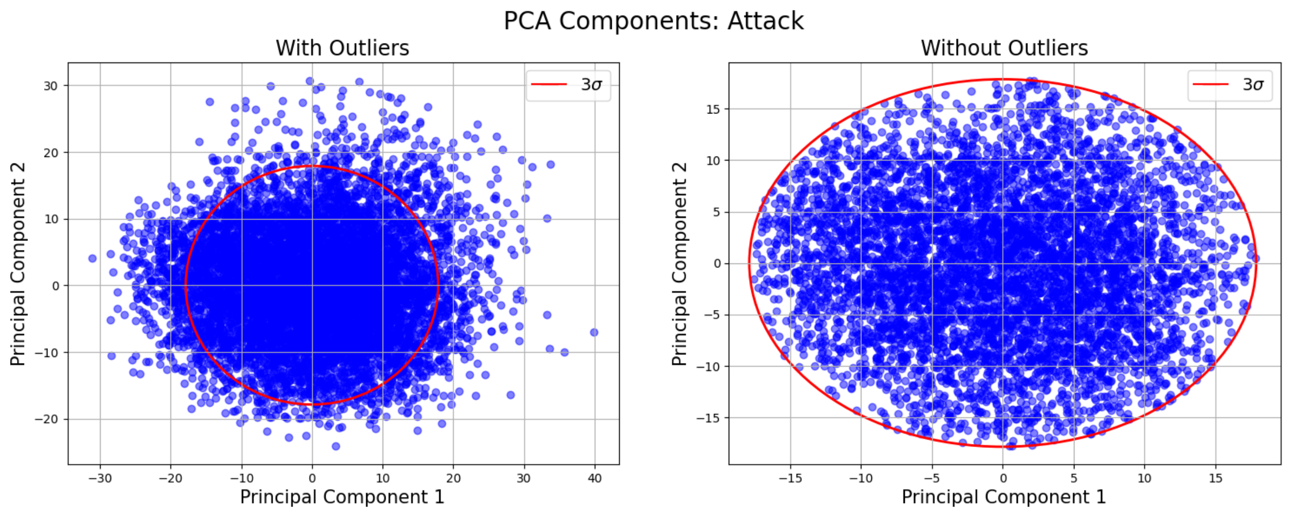
5 Future

The model successfully learnt to replicate every sequence in the training set with an average log loss below $\mathcal{L}_{Train} = 11.0$. However, despite tuning the hyperparameters and reducing the complexity of the dataset it was still heavily overfitting to the training set shown by $\mathcal{L}_{Test} = 14.3$. Therefore, the future of this project should focus on decreasing the overfitting so the model can be used to accurately predict unseen data.

L1 regularization is a technique that punishes the model for having large weights. This works by calculating the sum of all the absolute weights in the network and adding this to the cost function. L2 regularization is a similar method but uses the sum of the square of the weights which has a similar effect but is less likely to set weights to zero [27]. Both methods focus on reducing the weights in the network as this is a major cause of overfitting. This was already implemented using the *weight decay* argument in the optimiser. However, a more sophisticated



(a)



(b)

Figure 15: Graphs showing the principal components for the defending (a) and attacking (b) sequences before and after outliers have been removed. The 3σ level is shown by a red circle in each plot.

method can be implemented by directly calculating the sum in the cost function for weights in specific parts of the network that are causing the overfitting.

Another popular technique to reduce overfitting is ensemble learning. In general, this works by training multiple models and combining each output to reduce variance and bias associated with each model. Bagging ensemble learning works by splitting the dataset into subsets and training a model on each set. Boosting ensemble learning builds multiple models with a range of complexities [28].

Another potential issue could be the size of the dataset. Due to the amount of selection cuts used on the data, the number of sequences decreased from 30,000 to 10,000. As the relationship between the attacking and defending teams is highly complex, there is a strong possibility that this was simply not enough data. Therefore, a larger dataset could help to reduce the overfitting problem. This could be achieved by broadening the selection cuts to include more sequences or reducing the length of the sequences. This would not only increase the dataset but also decrease the complexity of the problem.

As spoken about in subsection 4.5, the varying order of the input players was an issue for the model. A potential fix for this would be to further sort the players by their exact position. This was not available in the dataset but could be calculated by the average positions over a sequence. However, this would be limited to attacking sequences with the same formation as a change in the number of players in each position group would shift the positions of each player in the input vector. A more rigid fix would be to change the structure of the input and output data. One suggestion is to quantise the pitch into 0.5 m x 0.5 m squares and assign each cell with a 1 if it included a player and a 0 otherwise. This would remove any dependence on the position of each coordinate in the input vector as swapping the positions of two players would not affect the input. However, to do this, a convolutional layer would have to be included in the network which would introduce more problems such as redefining the cost function.

6 Conclusion

In conclusion, the three main aims of the report were introduced and tested. The dataset was expanded by including a range of teams from a wider range of seasons. A new data filtering process which included a new Park-the-Bus filter was introduced which made sequences in the dataset more similar. Also, new training processes were introduced that were more suitable to the problem. On top of this, parameter initialisation was introduced to decrease convergence time and PCA was used to further refine the dataset. This combination of improvements meant that the model successfully learnt to replicate the training data with an average log loss below $\mathcal{L}_{Train} = 11.0$. This corresponded to an average separation between the expected and predicted positions below 3.5 m. However, despite hyperparameter tuning and the introduction of PCA, the model was still overfitting to the training set. This was shown by an average testing loss of approximately $\mathcal{L}_{Test} = 14.3$ which corresponded to an average separation of 19.9 m. Methods to reduce overfitting such as ensemble learning and redefining the structure of the input and output data were suggested as routes to take the project in the future.

References

[1] H Tabb. Investigating the effectiveness of Neural Networks in modelling a ‘Park the Bus’ football team. 2024.

[2] Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32, pages 1863–1871, Bejing, China, 22–24 Jun 2014. PMLR.

[3] Ilya Sutskever. *Training recurrent neural networks.* University of Toronto Toronto, ON, Canada, 2013.

[4] Prakash Venugopal. State-of-health estimation of li-ion batteries in electric vehicle using indrnn under variable load condition. *Energies*, 12(22):4338, 2019.

[5] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[7] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. Pmlr, 2013.

[8] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[9] Kamilya Smagulova and Alex Pappachen James. A survey on lstm memristive neural network architectures and applications. *The European Physical Journal Special Topics*, 228(10):2313–2324, 2019.

[10] Federico Landi, Lorenzo Baraldi, Marcella Cornia, and Rita Cucchiara. Working memory connections for lstm. *Neural Networks*, 144:334–341, 2021.

[11] Ralf C Staudemeyer and Eric Rothstein Morris. Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.

[12] Premier League, [redacted]. Accessed: 01/05/2024.

[13] Xue Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.

[14] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation. *International Journal of Intelligent Technologies and Applied Statistics*, 11(2):105–111, Jun 2018.

[15] Premier League seasons. https://fbref.com/en/comps/9/history/Premier-League-Seasons. Accessed: 01/05/2024.

[16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, page 2201–2206, New York, NY, USA, 2016. Association for Computing Machinery.

[17] [redacted]. Accessed: 01/05/2024.

[18] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[19] Gopal K Kanji. *100 statistical tests.* Sage, 2006.

[20] Roger J Barlow. *Statistics: a guide to the use of statistical methods in the physical sciences*, volume 29. John Wiley & Sons, 1993.

[21] Michael R Chernick. *The essentials of biostatistics for physicians, nurses, and clinicians.* John Wiley & Sons, 2011.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[23] Mohammad Mahdi Bejani and Mehdi Ghatee. A systematic review on overfitting control in shallow and deep neural networks. *Artificial Intelligence Review*, 54(8):6391–6438, 2021.

[24] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[25] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (pca). *Computers & Geosciences*, 19(3):303–342, 1993.

[26] Gary H McClelland. Nasty data: Unruly, ill-mannered observations can ruin your analysis. 2014.

[27] Andrew Y Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78, 2004.

[28] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms.* CRC press, 2012.

Appendix

Model Hyperparameters

Hyperparameter	Value
Learning Rate	0.001
Activation Function	Leaky ReLU
Hidden Layers	3
Hidden Dimensions	256
Train Test Split	80/20
Cycles per batch	1
Drop Out Coefficient	0.3
Batch Size	9,879
Sequence Length	45 frames
Initialisation	He
Optimiser	ADAM
Beta 1	0.95
Beta 2	0.999
EPS	10^{-8}
Weight Decay	0.005
Amsgrad	True

Table 3: A table showing the hyperparameters used in the final model with the top section showing overall model hyperparameters and the bottom section being specific to the ADAM optimiser.

Model Architecture

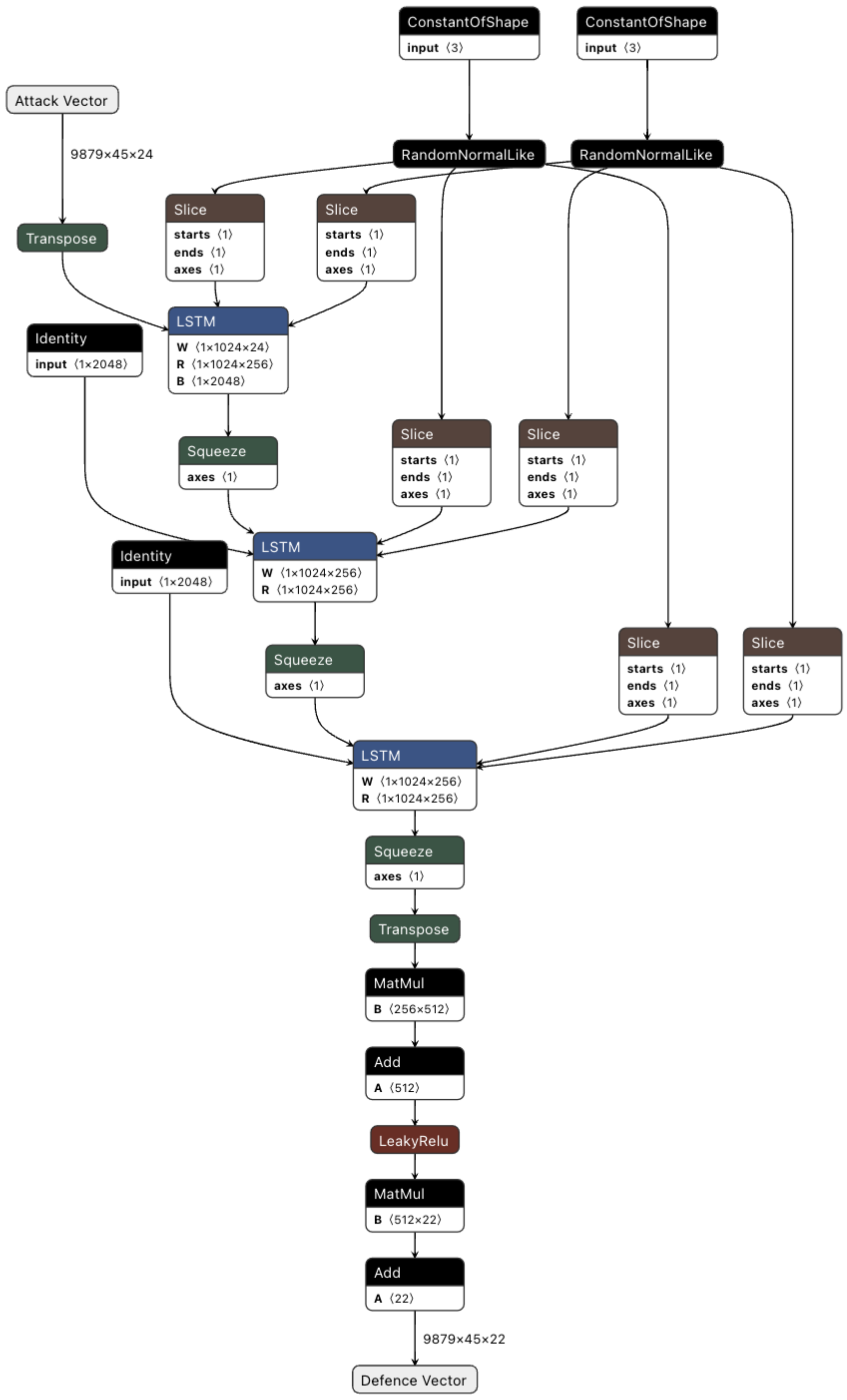


Figure 16: Diagram showing the structure of the model used in this project.