Investigating the Effectiveness of Neural Networks in Modelling a 'Park the Bus' Football Team

Harry Tabb

10635156

Mphys Project Report - Semester 1

School of Physics and Astronomy
The University of Manchester

January 2024

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

Abstract

This paper explores using different time series neural network architectures to predict defensive positions in a game of football. This is used to simulate the 'Park the Bus' defensive style used by [redacted] in the Premier League between 2016 and 2020. This predicted defensive positions to within 50 cm of the expected positions for the training data. However, when testing it predicted positions to within 11 m.

Contents

1 Introduction **1**

2 Theory **2**
2.1 Machine Learning . . . 2
2.2 Cost Function . . . 2
2.3 Neural Networks . . . 4
2.4 Optimisation and Backpropagation . . . 6
2.5 Recurrent Neural Networks . . . 8
2.6 Long Short Term Memory Networks . . . 9

3 Pre-Processing **10**
3.1 Pitch Geometry . . . 10
3.2 Data Structure . . . 10
3.3 Sequence Length . . . 11

4 Results **12**
4.1 Model 1: RNN . . . 12
4.2 Model 2: Basic LSTM . . . 13
4.3 Model 3: Enhanced LSTM . . . 15

5 Future **16**

6 Conclusion **17**

1 Introduction

In recent years, Artificial Intelligence (AI) has impacted various industries. For example in the medical field, DeepMind's AlphaFold, an AI built to understand protein structure has made significant strides in pharmaceutical production [1]. OpenAI's groundbreaking natural language model, GPT, has sparked a revolution by introducing intelligent virtual chatbots [2]. The automotive sector has also undergone a radical transformation courtesy of the implementation of AI into self-driving cars for companies such as Waymo and Tesla [3]. This rapid development is in part due to significant advancements in hardware technology. Not only have modern GPUs paved the way for faster and deeper computations but new sensor technology such as LIDAR has given access to a huge amount of new data.

This is particularly notable in the sports industry, especially in top football competitions such as the FIFA World Cup and the English Premier League. For example, high accuracy wearable GPS systems have been introduced for player and ball tracking, as well as high speed cameras in Semi Automated Offside Technology for modelling player positions [4]. This new hardware has enabled Premier League clubs to track player movements in a completely new manner which has paved the way for new data analysis techniques to revolutionise how the modern game is played.

This paper focuses on using this tracking data to simulate the 'Park the Bus' defence style used by [redacted] in the Premier League between 2016 and 2020. This team prioritised sitting back against teams that they knew they could not compete with on the ball. The idea behind this was to absorb pressure from the attacking team until an inevitable mistake was made which could be capitalised on with a counterattack. This led to low possession games from [redacted]; however, it was a successful strategy as even the best teams such as Manchester City and Liverpool struggled to break through the compact defensive structure. This is summarised in Table 2 by showing the percentage of total points dropped by 3 top teams when they had over 65% possession. This shows that despite having a large possession advantage, top teams can still struggle to break down teams that play this park the bus style and will drop points. Therefore, simulating how such teams play is key for breaking through these defences. For example this could be used to see how [redacted] would theoretically defend and weaknesses in their structure against specific attacks could be investigated and exploited before the game day.

Team	2022/23	2021/22	2020/21	2019/20
Manchester City	68%	71%	46%	70%
Liverpool	49%	32%	49%	80%
Chelsea	30%	43%	38%	50%

Table 1: A table showing the percentage of total points that were lost against teams that had less than 35% possession for 3 teams that were consistently in the top 5 average possession from the 2019/20 season to the 2022/23 season in the English Premier League [5].

Therefore, using player and ball tracking data provided by Opta, this paper investigates how to best predict defensive formations given the positions of the attacking players and the ball. This aims to simulate how [redacted] would theoretically defend against a new style of attack. The proposed method to do this is to experiment with different time series neural networks to investigate which style most accurately reproduces and predicts defensive structures. A game of football is far too complicated to accurately predict whole games at a time. Therefore, games were broken up into shorter sequences of well structured defensive play as these are more manageable for a neural network to model.

2 Theory

2.1 Machine Learning

Machine learning is a subset of Artificial Intelligence that focuses on building algorithms and statistical models that make predictions from patterns in data without specific manual instruction. This broad definition is then traditionally broken down into three further divisions, supervised, unsupervised and reinforcement learning.

In supervised machine learning, the datasets that are used to train the model have labels, it is this label that the model tries to predict based on other features in the data. This is done by using a cost function that quantifies how close the model prediction is to the expected output. The parameters are then tweaked based on the cost to find patterns in the data to improve the prediction. After the training process, the model is given unseen data and makes a prediction for the label based on the other features. This is the testing process. Supervised learning is normally used in regression problems with continuous labels where the model tries to predict values such as coefficients of a polynomial. Or classification problems with discrete labels where the model tries to predict a category to assign to the input such as sorting images into categories.

Unsupervised machine learning is fundamentally different as the model is trained to find patterns in an unlabelled dataset. As the input data does not have a target output or label, there is no cost function so measuring the performance is not as straightforward. Unsupervised learning is often used for clustering parts of a dataset together based on common features or reducing the number of dimensions in a dataset whilst preserving the most important features. Both of these applications are normally used as a precursor to further analysis, so the performance of the model can be evaluated based on the outcome of the subsequent tasks.

Reinforcement learning involves an agent that takes actions between states in an environment. Therefore, the model is not given a dataset to learn from, instead, the agent moves around the state space and is given penalties and rewards. This is analogous to the cost function in supervised machine learning and helps the model to learn which actions to take in a given state. This is often used in robotics and optimisation problems as observing and interacting with an environment is key for decision making.

This paper uses supervised machine learning to perform a regression to predict the positions of the defending team given the positions of the ball and the attacking players. A specific type of supervised machine learning called neural networks is used because a simple regression model would not be able to capture the complex patterns between the attacking and defending teams.

2.2 Cost Function

In machine learning, a cost function (or loss function) is required to quantify the error between the model's prediction and the expected training data. The goal of the training process is to minimise this error. In the context of this paper, this corresponds to minimising the difference between the model's predicted positions of the 11 defending players, and their true positions from the tracking data. To avoid complications relating to the order of the output and expected arrays, the Hungarian algorithm is used. Instead of calculating the loss between the array elements pairwise, the optimum pairings are found by calculating a cost matrix and finding the minimum of each row.

Initially, a mean squared error cost, C_{MSE}, was used for each frame which is defined as

$$C_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(r_i^{\text{exp}} - r_i^{\text{pred}})^2 \tag{1}$$

where N is the number of players in the frame and r_i^{exp} and r_i^{pred} are the positions of the expected and predicted players respectively.

The cost of a sequence was then calculated by taking the mean average of the cost of each frame in the sequence. The mean average of the sequence costs in a batch was then calculated for the cost of a batch.

Cost functions are unlikely to reach zero in finite time, they either asymptote to a non-zero value or get stuck in a local minimum. This is particularly common in high dimensional space, therefore, as the cost is a function of 22 variables, it was expected that the model would not predict the exact positions of the defending players. In the context of the model's predicted player position, a non-zero cost calculated solely from the distance between the expectation and predicted positions results in a locus of points around the true position of the player. The predicted position can be at any point on this locus; however, from a football perspective, every point on this locus results in a very different defending position as seen in Figure 1. Therefore, an angular cost around this circle is also minimised to predict the best defending position on this circle. This polar angle, ϕ, is defined from the x axis and $-180 < \phi \leq 180$. Mathematically, this is given by

$$\phi = \text{arctan2}(\frac{y^{\text{pred}} - y^{\text{exp}}}{x^{\text{pred}} - x^{\text{exp}}}) \tag{2}$$

where $(x^{\text{pred}}, y^{\text{pred}})$ and $(x^{\text{exp}}, y^{\text{exp}})$ are the predicted and expected coordinates of the player respectively and . The angular cost, C_ϕ, is then defined as

$$C_\phi = |\phi - \psi| \tag{3}$$

where ψ is given by

$$\psi = \text{arctan2}(\frac{y^{\text{exp}} - y^{\text{goal}}}{|x^{\text{goal}} - x^{\text{exp}}|}) \tag{4}$$

where $(x^{\text{goal}}, y^{\text{goal}})$ are the coordinates of the centre of the defending goal. This is the angle between the centre of the goal and the true position of the defending player measured from the x axis, as seen in Figure 1. This is negative above the x axis and positive below. Therefore, the angular cost is a minimum when the predicted position is between the expected position and the centre of the goal (goal side, shown by point A in Figure 1) and increases counter clockwise in around the circle to its maximum value when the predicted position is on the opposite side of the circle (point B in Figure 1). The angular cost is defined over a semi-circle such that the cost of points C and D in Figure 1 will have the same absolute value so that the function can decrease both in both directions to improve the efficiency of the training process.

When the distance between the predicted and expected position is large, the angular cost is not as important as the distance cost. Therefore, for the total cost, C, a constant is included such that the angular cost is comparable to the distance cost when the distance between the predicted and expected is of the order of 1 meter. So, the final definition of the cost function is given by

$$C = C_{\text{MSE}} + 50C_\phi. \tag{5}$$

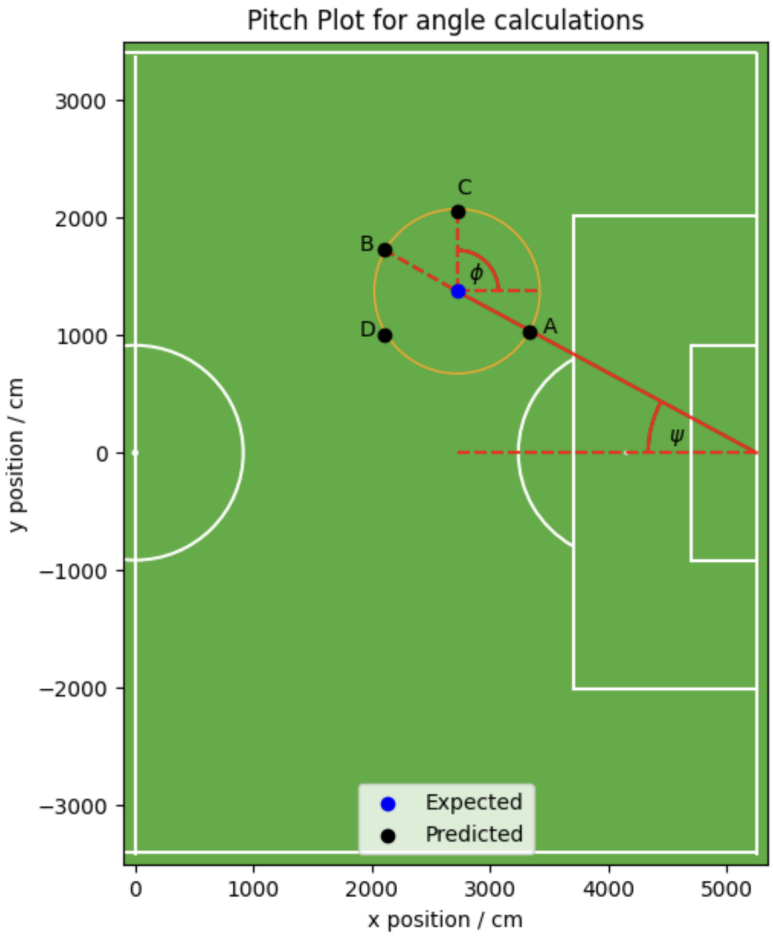


Figure 1: A pitch plot showing the angles used in defining the angular cost function. ψ is the offset angle and ϕ is the angle around the loci of predicted positions. Both are measured from the x axis. Point A is the predicted position that minimises the angular cost, Point B is the predicted position that maximises the angular cost. Points C and D have the same absolute angular cost but D has a negative cost and C has a positive cost.

2.3 Neural Networks

Neural networks (NNs) are a subdivision of machine learning that is inspired by the way biological nervous systems process information [6]. They are composed of 2 main components: neurons and the connections between the neurons. The neurons are organised into layers that are stacked on top of each other and joined by the connections as seen in Figure 2. These layers are categorised into three types: the input layer, hidden layers and the output layer. This section explains the role of each of these layers in how neural networks make predictions. The input layer is the only layer that directly interacts with the input data. Each neuron has an activation that corresponds to the value in each dimension in the input data. Similarly, the output layer is responsible for the model output so it has a neuron for each dimension of the output data with an activation corresponding to the value in the output. In the context of this paper, each neuron in the input layer has an activation given by the attacking player's coordinates, and the activations of the neurons in the output layer correspond to the defending player's coordinates. The hidden layers lie between the input and output layers and are responsible for the model calculations. The number of neurons in each hidden layer and the number of hidden layers are hyperparameters that are chosen to best match the complexity of the particular problem. The activation of each neuron in the hidden layers, a_i, is calculated from a linear combination of the activation of the neurons in the preceding layer. Mathematically, this

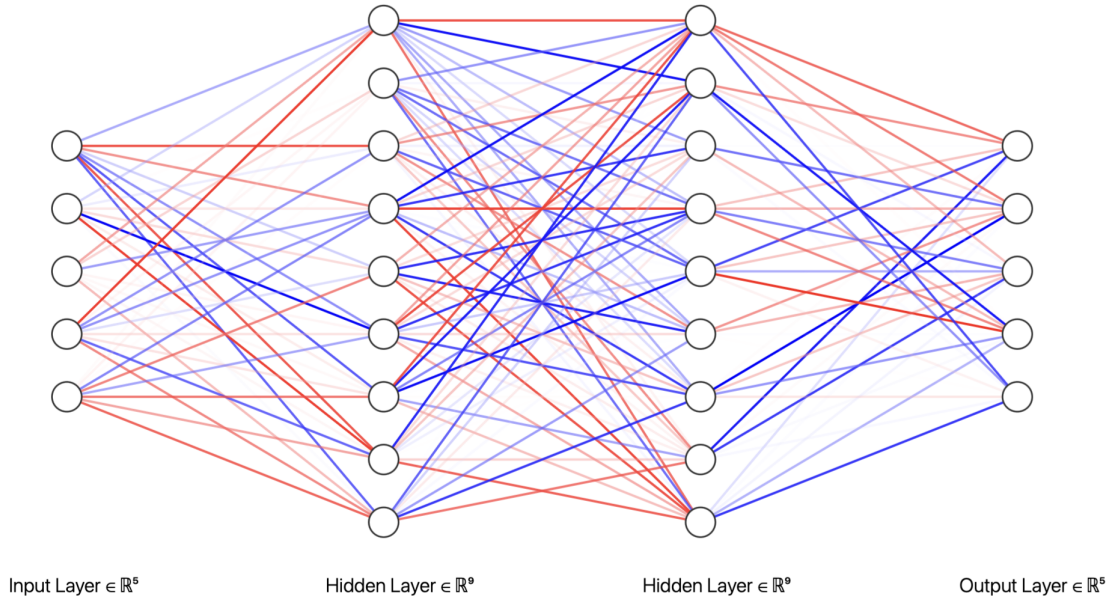


Figure 2: A schematic diagram of a basic neural network with 2 hidden layers, an input layer, and an output layer. Each layer is labeled with its type and the number of neurons in the layer. Neurons are shown by white circles and connections are shown by straight lines. Red lines correspond to negative weightings and blue lines correspond to positive weightings. The opacity of the line represents the magnitude of the weighting.

is given by

$$a_i = \Sigma_j w_{ij} a_j + b_i \tag{6}$$

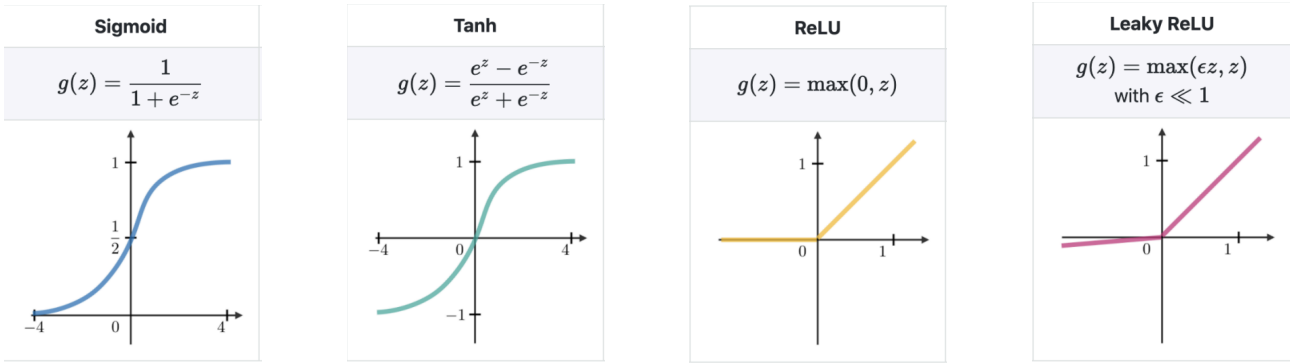
where w_{ij} is the weight between neuron j in the previous layer and neuron i in the current layer, a_j is the activation of neuron j in the previous layer and b_i is the bias of neuron i in the current layer. The weights and biases are model parameters that are finetuned by comparing the output of the model for a given set of parameters to the expected output using a cost function. This cost is then minimised by adjusting the parameters over multiple iterations of training until it reaches zero or asymptotes to a finite value. The weight between neurons is a measure of the strength of the connection between features in each layer. The bias is added to the sum to introduce a threshold for how high the weighted sum needs to be for the neuron to be activated. This shifts the centre of the activation such that it is centred at the bias instead of zero. Therefore, it allows the model to learn patterns that are offset from the origin which makes neural networks more flexible to a range of datasets.

Most neural networks also include an activation function, this is applied to the linear sum of activations to introduce non-linearity. This allows the model to be more flexible in learning more complex relationships, as it is not limited to linear relationships. The choice of activation function depends on the task and properties of the network as each function transforms the sum in different ways. The most common examples are the Sigmoid, Tanh, Rectified Linear Unit (ReLU) and Leaky ReLU functions, these are shown in Figure 3. The Sigmoid (a) transforms the activation between 0 and 1, therefore it is most commonly used in binary classification problems. The Tanh function (b) is very similar to the Sigmoid but is defined between -1 and 1. The ReLU function (c) returns 0 for negative activations and returns the input otherwise. The Leaky ReLU function (d) is similar to the ReLU but has a small positive gradient for negative activations. The choice of activation is dependent on the specific task of the model, however, the Leaky ReLU is used as the most common choice. The normal ReLU is extremely simple which leads to fast computations, it also helps negate the vanishing gradient problem

which is common with the Sigmoid and Tanh functions. However, it does lead to the dead neuron problem. This happens when a neuron consistently receives a negative input so it will become inactive as the ReLU always returns zero for negative activations [7]. However, the leaky ReLU has a small gradient for negative inputs which prevents neurons from dying which leads to better model predictions whilst maintaining the fast computation speed of the ReLU. Therefore, the Leaky ReLU is often the best choice of activation function in the hidden layers. With the activation function, σ, Equation 6 becomes

$$a_i = \sigma(\Sigma_j w_{ij} a_j + b_i) \tag{7}$$

which is the final equation used to calculate the activation of each neuron in the hidden and output layers.



(a) Sigmoid (b) Tanh (c) ReLU (d) Leaky ReLU

Figure 3: A diagram showing the 4 most common activation functions used in neural networks [8].

2.4 Optimisation and Backpropagation

During the first iteration of the training process, the model parameters need to be set before the optimisation process can start. There are different methods to choose these parameters, but the most common choices are zero and random initialisation. Zero initialisation sets all the parameters to zero, this is simple but often leads to slow convergence and symmetry issues in deep networks. Random initialisation sets each parameter randomly from a normal distribution, this is more complex but breaks any symmetry and prevents neurons from learning the same features. The standard deviation of the normal distribution further defines the type of initialisation, this is often chosen to match the activation function. For example, a variance of $\frac{2}{n_{\text{in}}+n_{\text{out}}}$ (Xavier initialisation) is used with a tanh function [9] whereas a variance of $\frac{2}{n_{\text{in}}}$ (He initialisation) is used with a ReLU function [10], where n_{in} and n_{out} are the number of input and output neurons respectively.

Once the initial parameters have been defined, the accuracy of the model's output is quantified by passing it through a cost function. This compares the output to the expected data and returns a single value for how close the prediction was. The cost function can be easily visualised with two model parameters as shown in Figure 4, each pair of parameter values has a well-defined cost. By visualising the cost function in 3 dimensions it can be seen that the cost is minimised by calculating the gradient of the cost function with respect to the model parameters and changing the parameters along this direction. This is called gradient descent. Finding these gradients is known as backpropagation as they are calculated from the final layer of the network towards the first. For each iteration of training, the new value of each parameter, θ_i^*, is then

calculated using

$$\theta_i^* = \theta_i - l\frac{\partial}{\partial\theta_i}C \tag{8}$$

where θ_i is the old value of the parameter, l is a hyperparameter called the learning rate which controls the step size of the gradient descent. For most types of optimisation algorithm, the learning rate is fixed for all parameters. This process is then repeated until the cost asymptotes or falls below a certain value.

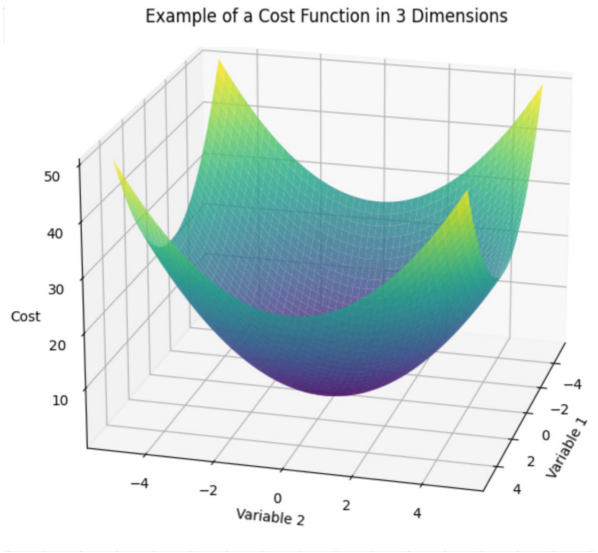


Figure 4: A graph showing an example of a cost function in 3 dimensions. The two variables that affect the cost function are on the x and y axis. The cost is on the z axis.

Although difficult to visualise in a higher dimensional parameter space, the same method still applies. However, with more model parameters, the cost function will not be as simple as in Figure 4, it will most likely not be a simple parabola resulting in many local minima. Therefore, an analytical gradient descent like described above is not always the best choice as it is computationally slow in high dimensions, can easily get stuck in local minima and is very sensitive to the choice of learning rate. Therefore, other optimisation algorithms such as stochastic gradient descent (SGD) and adaptive moment estimation (ADAM) are used to avoid these problems.

In SGD, each iteration of training randomly selects a 'mini batch' of the training data to perform the analytical gradient descent on. This improves computational efficiency as less data is being used, but the whole data set is represented as random batches are selected every iteration. These mini batches also help the optimiser to avoid local minima as different data is used at each iteration. However, due to the stochasticity, this inevitably leads to noisier costs, and thus a slower convergence to the optimum solution. By including a further parameter called the momentum into SGD, the noisy costs and slow convergence can be avoided. This incorporates a moving average of past gradients into the calculation for the current update. However, even with including the momentum, it can still be difficult to find the best learning rate for the particular problem. If l is too small, the convergence will be very slow. If it is too big, it will likely step over minima and never reach an optimum value. This is why ADAM is normally a better choice of optimiser.

The fundamental difference in ADAM is it has a different learning rate for each model parameter which is constantly updated throughout the training process. This is done by calculating the first moment for each parameter, this is an exponentially weighted mean of all past gradients weighted to place emphasis on more recent gradients. This defines the direction the parameters

should be moved. The second moment (variance) is also calculated to track the fluctuations of past gradient fluctuations. These moments are combined to calculate an adaptive learning rate for each parameter, parameters with large second moments (consistently large gradients) will be assigned low learning rates to avoid overshooting a minimum, and parameters with small second moments are assigned large learning rates to move towards minima faster. ADAM is much more efficient due to these adaptive learning rates, and it also helps to avoid local minima. As the learning rates are calculated by the optimiser, the initial choice of the learning rate is less important, so it is much easier to use, giving more reliable results.

2.5 Recurrent Neural Networks

As this paper focuses on finding patterns in time series data, a simple neural network that only finds patterns in static data is not sufficient. A Recurrent Neural Network (RNN) is a special type of Neural Network that can find patterns and make predictions on time series data, the current prediction is dependent on the current input and all inputs before it. Therefore, they are powerful for speech recognition, time series prediction and language translation. The key difference between the architecture of a normal Neural Network and an RNN is the addition of a feedback loop as shown in Figure 5. Instead of data being passed in a straight line through each layer of the network, after each hidden layer there is a feedback loop with its own weighting that is added to the linear sum for the next input. Therefore, Equation 7 now becomes

$$a_i = \sigma(\Sigma_j w_{ij} a_j + b_i + w j_{i-1}) \tag{9}$$

where w is the weight of the feedback loop and j_{i-1} is the hidden output of the previous time step. The weighting, w is a measure of the importance of the previous inputs in the current output, however, this is constant for each time step. This is a clear limitation of an RNN as events at the start of the sequence can have a large effect on the output at the end of the sequence. For long sequences, this may not be appropriate if events at the start don't correlate to outputs at the end.

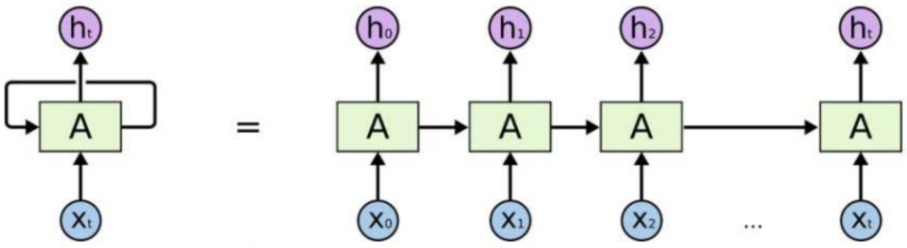


Figure 5: A schematic diagram of the architecture of a Recurrent Neural Network. The left hand side shows the feedback loop going back into the network. The right hand side shows this unrolled for clarity [11].

A bigger issue is the vanishing and exploding gradient problem. If the weighting of the feedback loop, w is larger than 1, then for long sequences, the final output will be multiplied by this weighting as many times as the length of the sequence. Therefore, the gradient will get large very quickly. This means that in the optimisation process, it will be impossible to take small steps in the cost function and the minimum will likely be missed due to the large step size. The same problem occurs if w is less than 1, this is the vanishing gradient problem. For long sequences, the gradient will tend to zero meaning the steps across the cost function will also tend to zero and the model will never converge. This is a big limitation of RNN's, the most popular fix is to use a Long Short Term Memory Neural Network (LSTM) instead.

2.6 Long Short Term Memory Networks

An LSTM is a specific type of RNN that fixes the exploding and vanishing gradients problem and is capable of learning long term dependencies. They were first introduced by Hochreiter and Schmidhuber in 1997, but they have been refined by many other people since [12]. Instead of repeating a single layer like an RNN, an LSTM repeats 4 intertwined layers for each frame in the sequence. The long term memory comes from the cell state which is shown by the horizontal line across the top of Figure 6. This runs down the whole network and does not have any weights or biases, its value is updated solely by the gates. The first gate is the forget gate, this enables the network to selectively forget long term information that is no longer relevant. This is done by calculating what percentage of the previous long-term memory, C_{t-1} should be kept based on the previous hidden layer, h_{t-1}, and the current input, x_t. The weights and biases corresponding to h_{t-1} and x_t are tuned using backpropagation during in the training process and a sigmoid activation function returns a value between 0 and 1 which corresponds to the percentage that is kept.

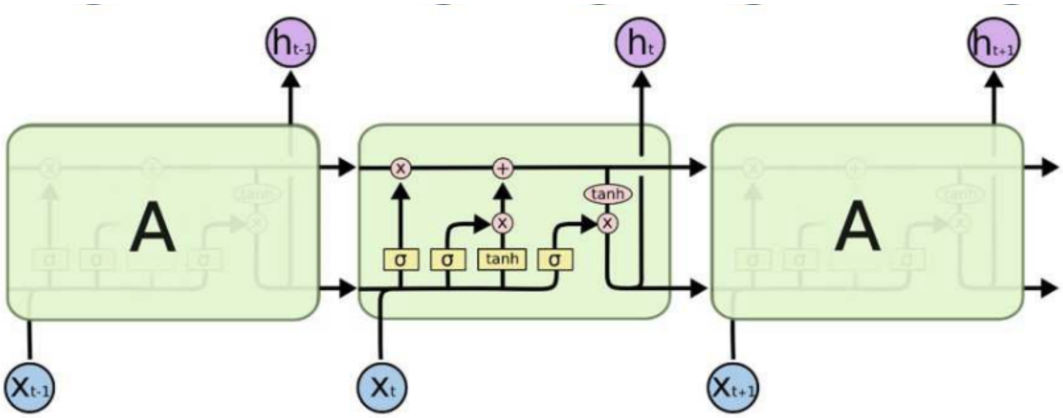


Figure 6: A schematic diagram showing the architecture of a Long Short Term Memory network. Each cell of the model is shown in a green box with the inputs in blue circles and hidden outputs in purple circles. The gates different gates are shown in the cell by a combination of the relevant activation functions [11].

The next gate is the input gate. This decides what parts of the current input are going to be added to the long-term memory based on the current input and the previous hidden output. This is achieved by a combination of sigmoid and tanh functions.

The final gate is the output gate. This decides which parts of the long-term memory are going to influence the output at the current time step. Similarly to the input gate, this also uses a combination of tanh and sigmoid functions. As with normal Neural Networks, the weights and biases corresponding to the connections between the nodes in each gate are fine-tuned using backpropagation as described in subsection 2.4.

The introduction of these gates offer a solution for the vanishing and exploding gradients problem. The idea of repeatedly applying the same weight when calculating gradients for frames far into the sequence as in an RNN no longer applies to LSTMs. The gate parameters are different for each time step and the network can choose to forget information from far back in the sequence. Therefore, the exploding and vanishing gradient problem and not being able to predict long term dependencies in an RNN are both fixed by using an LSTM.

3 Pre-Processing

3.1 Pitch Geometry

For the extent of this paper, the geometry of the football pitch is defined as shown in Figure 7. The origin of the coordinate system is at the centre spot with the x axis running between the goals and the y axis between the side lines. As each Premier League stadium has different dimensions, the pitch dimensions needed to be normalised so they were constant for every game. As [redacted] is the focus of the paper, [redacted] was used to define the normalised pitch dimensions of 105 m in length and 68 m in width [13].

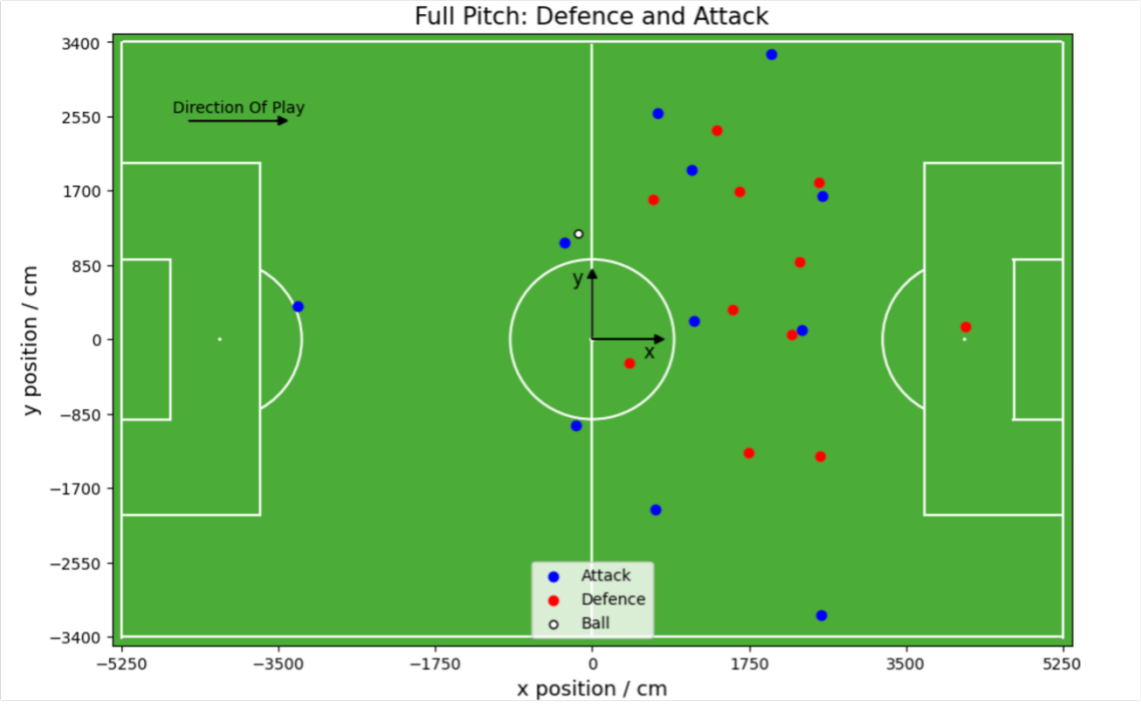


Figure 7: A figure showing an example of a pitch plot for a given frame. The defending players are plotted in red and the attacking players are plotted in blue. The coordinate system is centred at the centre spot of the pitch with the x direction pointing along the length of the pitch and the y direction pointing along the width. The direction of play is defined to be in the positive x direction.

Opta defines their tracking data such that the home team attacks in the positive x direction. However, as [redacted] defensive structure is the primary focus of this paper, this was redefined such that the attacking team was always playing in the positive x direction, as shown in Figure 7.

3.2 Data Structure

The premise of this paper is to create a neural network that will predict the positions of the defending players given the positions of the attacking players. Therefore, the model took the x and y coordinates of the 11 attacking players, as well as the x, y, and z coordinates of the ball as its input. As RNNs and LSTMs can only process 1-dimensional data, these coordinates were inputted into the model as a 1-dimensional vector. The model then outputs the 11 coordinates of the players in the defending team as a 1-dimensional vector.

As LSTMs are used to make predictions on time series data, the frames from the tracking data needed to be grouped into sequences. Each sequence was 9 seconds long which equated to 45

frames as the tracking data was recorded at 5fps. These sequences were extracted from the raw tracking data by filtering for phases of play for which the defending team was out of possession and had 11 players in their own half. Other criteria also needed to be met for a sequence to be valid. The ball needed to stay in play for the whole sequence and the attacking team had to be in constant possession. This ensured a constant phase of play with no interruptions. The half way line cut off was chosen as it was found that if a defending player was beyond this point, they tended to play more of a 'pressing' role as opposed to a 'park the bus' role. It was found that pushing this selection cut further into the defending half increased the amount of 'park the bus' frames; however, it also removed more data. Therefore, the halfway line was chosen as a large number of frames were still 'park the bus' (high purity) and it still left enough data to accurately capture the complex relationships between the attacking and defending team and to prevent over fitting.

Once the sequences had been extracted, they were sorted into batches as when training the model multiple sequences can be trained at once for efficiency. The batch size is a hyperparameter that is chosen to suit the amount of data and the computational power. These batches were then split into a training set and a testing set. The amount of data in the training set and the amount in the testing set, the train-test split, is another hyperparameter that needs to be tuned to avoid over fitting.

3.3 Sequence Length

The aim of this section is to calculate how long each sequence of play should be. In order to do this, games were modelled as a Markov chain, this is a stochastic process that moves among a set of states [14]. As the transition probabilities do not vary in time and the Opta data was recorded at a discrete 5 frames per second, this definition is expanded to a discrete time, homogenous Markov chain. The time taken for the Markov chain to reach its stationary state was then defined as the maximum sequence length.

To model this as a Markov chain, a state space containing all possible configurations of the system needed to be defined. For this, the pitch was quantised into 1 m^2 squares such that the state of the process at a time, t, was given by the box containing the ball at that time. The transition matrix of a Markov chain is an $n \times n$ stochastic matrix given by

$$Q = \begin{pmatrix} q_{11} & \cdots & q_{1n} \\ \vdots & \ddots & \vdots \\ q_{m1} & \cdots & q_{mn} \end{pmatrix} \tag{10}$$

where n is the number of states, and q_{mn} is the probability of transition from state n to state m in a time step Δt. For each time step, the position of the ball at the current and the previous time step was recorded and each transition was normalised by the total number of transitions out of that state to calculate the transition probabilities between the two states. Therefore, the state of the system at a time t, P_t, is calculated by

$$P_t = QP_{t-1} \tag{11}$$

where P_{t-1} is the state of the system at a time $t-1$. The stationary state of a Markov chain, P_{st} is defined as

$$P_{\text{st}} = QP_{\text{st}} = Q^\tau P_t \tag{12}$$

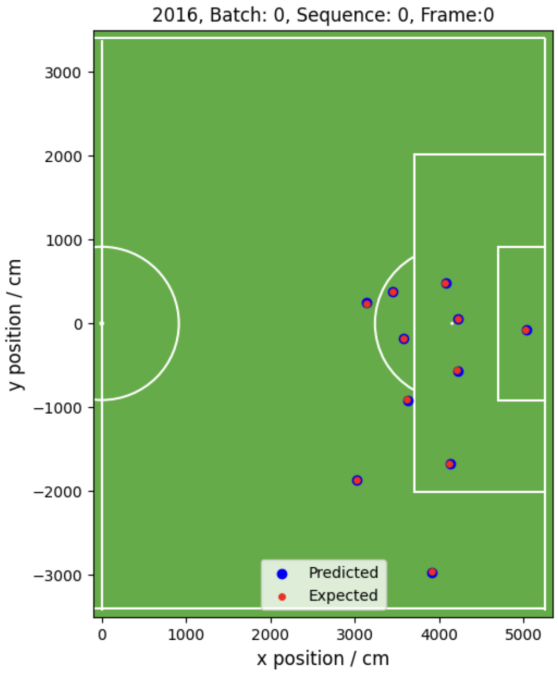
where τ is the number of time steps to go from P_t to the stationary state which is the state of the system that will not evolve any further when acted upon by the transition matrix. So this time, τ, was defined to be the length of the sequence as after this the state is stationary. This was found by multiplying Q by itself until components changed less than 5%. This gave $\tau = 47$ frames which corresponds to 9.4 s and was rounded down to 9 s for simplicity.

4 Results

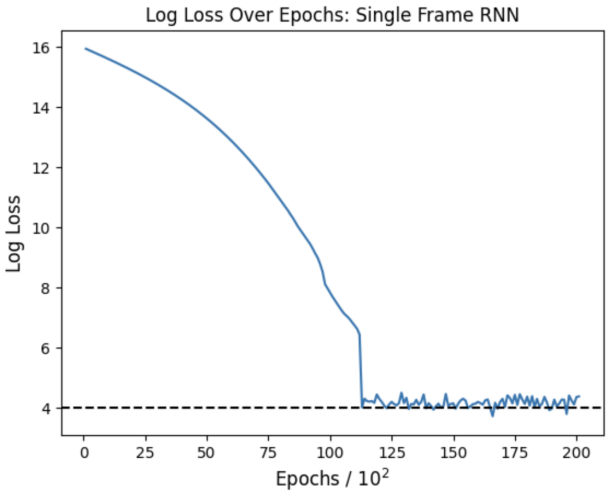
Over the course of the project, a range of different types of models and architectures were used. In this section, the outputs of each model in the training process are compared. Each model used ADAM as its optimiser as this consistently provided the best results as described in subsection 2.4. All other hyperparameters for the final model are listed in the appendix.

4.1 Model 1: RNN

The first, most basic model that was trained was a simple RNN with 1 hidden layer. As shown in Figure 9, this showed promise for a singular frame. It almost perfectly replicated the positions of all 11 players. The log loss function has an asymptote at approximately $\log(\text{loss}) = 4$ which was reached in under 20000 iterations of training. A log loss plot was chosen to display this data as $0 < \text{loss} < 10^6$. A log loss of 4 corresponds to an average separation between the expected and predicted below 10 cm when also accounting for the angle. From here this is defined as an optimum loss, anything below a log loss of 9.5 is considered good as this corresponds to an average distance of approximately 100cm. However, this did not generalise well for a whole sequence. Such a simple RNN struggled to capture any patterns over a long sequence. The results are shown in The predicted positions were far from the expected positions across the whole sequence. This is quantified by the loss asymptoting at $\log(\text{loss}) = 11.8$ after 10000 iterations, it would not have been computationally favourable to train for any longer than this as the loss had reached an asymptote. Despite the expected positions moving throughout the sequence, the model only predicted two different points over the 45 frames. This is clear evidence that the model is struggling to predict any time dependency, and it is just predicting an average position for each player. Deeper RNN's were experimented with (more hidden layers and more hidden dimensions), however, this just increased the convergence time with little difference in predictions. Therefore, the next logical step was to try an LSTM to predict a whole sequence.

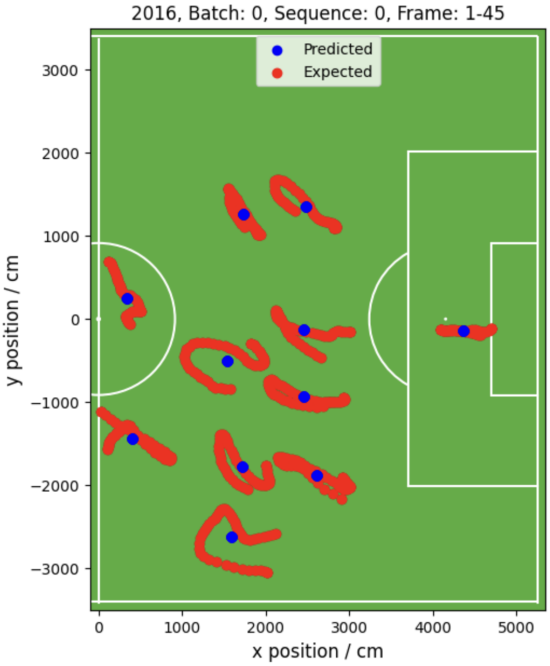


(a) Pitch Plot for 2016, Batch 0, Sequence 0, Frame 0.

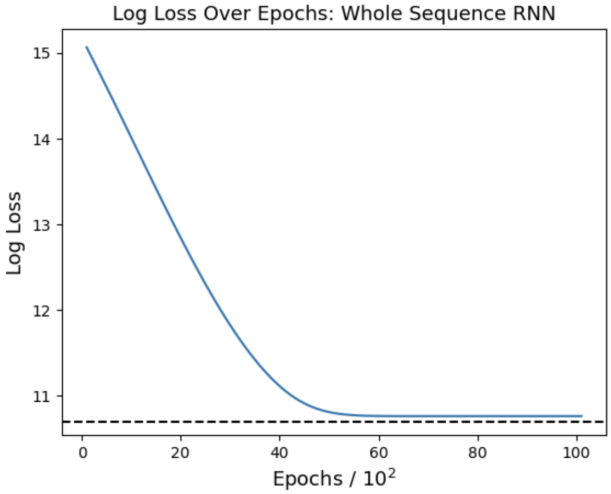


(b) Log Loss against Epochs.

Figure 8: Diagrams showing the pitch plot of the expected and predicted positions of the defending players and a log loss against epochs graph for a single frame RNN.



(a) Pitch Plot for 2016, Batch 0, Sequence 0.



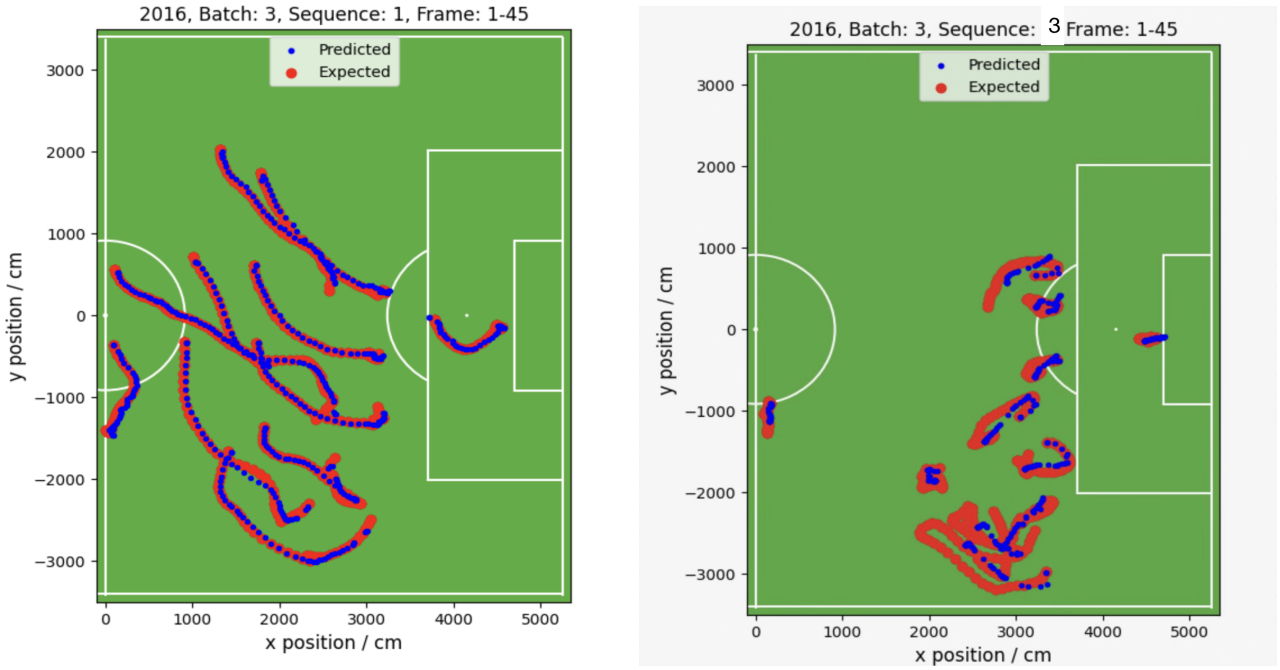
(b) Log Loss against Epochs.

Figure 9: Diagrams showing the pitch plot over a whole sequence of the expected and predicted positions of the defending players and a log loss against epochs graph for an RNN.

4.2 Model 2: Basic LSTM

The second model was a basic LSTM. It had exactly the same structure as model 1 but the two RNN layers were replaced by LSTM layers. This still accurately predicted a single frame, but

more importantly, it started to predict time dependencies throughout the sequence. Figure 10 shows the expected and predicted paths two different sequences in the same game. For the second sequence, some player paths were tracked well while others were very erratic. It was these paths that drove the cost up. It is easily seen from Figure 11 that this model predicted positions well for some sequences but poorly for others. The good sequences such as sequence 1, reached log(loss) = 8 and the bad sequences such as sequence 3 reached log(loss) = 11 which is not below the threshold for a good prediction. This was an improvement from the first model as full sequences were reproduced to a high accuracy. However, the model had to be able to reproduce sequences across a whole game, and eventually a whole season. To do this the model needed to be much deeper, therefore the next step was to increase the complexity of the LSTM.



(a) Pitch Plot for all frames in 2016, Batch 3, Sequence 1.

(b) Pitch Plot for all frames in 2016, Batch 3, Sequence 3.

Figure 10: Diagrams showing the pitch plot of the expected and predicted positions of the defending players for a whole sequence LSTM for two different sequences.

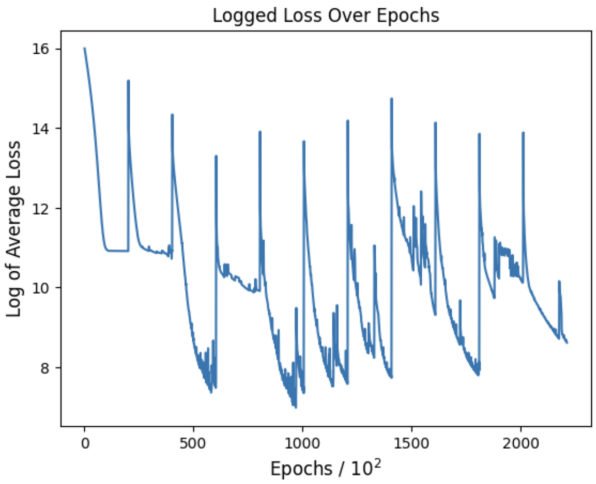
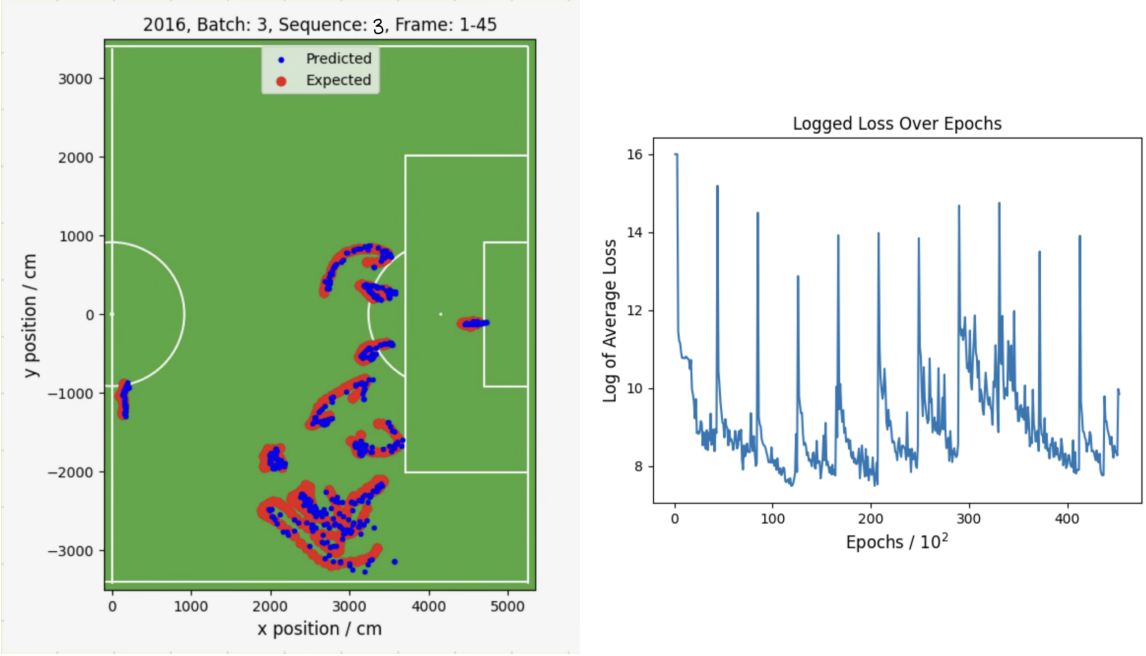


Figure 11: A graph showing the log loss over epochs for all sequences in 2016, Batch 3 from the LSTM.

4.3 Model 3: Enhanced LSTM

The final model was an enhanced LSTM, which meant using 2 LSTM layers with a drop out layer, 2 fully connected layers and a leaky ReLU activation function. This new model was trained on the same game as in subsection 4.2. The results of this are shown in Figure 12. The bad sequence from model 2 (sequence 3) which previously had an asymptote at log(loss) = 11 now has log(loss) = 7.5 which is below the threshold of a good prediction. The pitch plot for the same sequence is also shown in Figure 12. Most player paths are much more accurately predicted over the whole sequence. However, the 3 player paths at the bottom still have slightly messy predictions. This is likely due to how close the player paths are. The worst sequence in the game (the highest loss) has a log(loss) = 9 which is still below the threshold. Therefore, model 3 successfully reproduces a whole game of sequences. However, it struggled to generalise for a whole season of games. Ontop of this, as seen in Figure 13, when testing the model on unseen data, it is heavily overfitting to the training data. The test loss has a minimum of log(loss) = 13.5 which corresponds to an average distance of 11 m between the predicted and expected positions. This is clearly not a good prediction. This means that the model learnt the training data too well and captured noise and random fluctuations rather than the underlying patterns that are consistent between the training and testing data.



(a) Pitch Plot for all frames in 2016, Batch 3, Sequence 3.

(b) Log loss graph over epochs for all sequences in 2016, Batch 3 from the enhanced LSTM.

Figure 12: Diagrams showing the pitch plot of the expected and predicted positions of the defending players for a whole sequence LSTM for two different sequences.

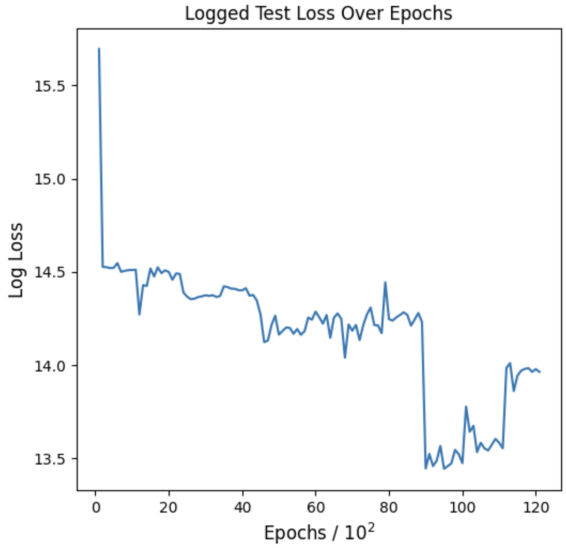


Figure 13: A graph showing the logged test loss over epochs for 2016, Batch 3 from the enhanced LSTM.

5 Future

Despite being able to reproduce sequences from a singular game, the model cannot make predictions to unseen data at this point. The aim of building a neural network is to be able to predict the output of unseen data. Therefore, reducing this overfitting and increasing the quality of the outputs is a key next step for the project. Methods for doing this include adding random noise to the input data and adding early stopping.

The model can accurately reproduce the training data for a single game; however, it struggles to do this consistently over a whole season. To fix this, the data needs to be easier for the model to learn. One way to do this is by applying more filters in the data processing stage. Examples of what could be done include, implementing more successful filters for removing complicated phases of play that are hard to learn such as corners and free kicks. As these can be quite difficult to filter mathematically unsupervised machine learning techniques such as clustering algorithms could be implemented to sort defensive structures. Another way to help the model learn would be to change the structure of the data completely. Neural networks tend to be able to find patterns in spatial relationships from a matrix structure better than numerical coordinates [redacted]. Therefore, changing the data to be in a sparse matrix format with a 1 representing the position of a player on the pitch and a 0 representing an empty grid cell could help the model learn patterns better. However, an LSTM cannot handle this type of input as it can only take a 1 dimensional input. However, adding a convolutional layer before the LSTM layer turns the model into a convolutional LSTM. This means that the model will be able to take 2 dimensional inputs and hopefully will be able to find spatial relationships between players better and faster thus increasing the quality of output and reducing over fitting.

Also, at the moment, only sequences that are exactly 45 frames long are used in the training process as LSTMs cannot handle variable sequence lengths without padding. This greatly decreases the amount of data that is extracted from a single game so ideally, this needs to be increased. Therefore, by incorporating padding techniques into the data processing stage, the amount of data being filtered out of the dataset will increase and this will decrease over fitting and increase how much the model learns.

6 Conclusion

The aim of the project was to build a machine learning model that successfully reproduced how [redacted] defended in a range of Premier League seasons between 2016 and 2020. To do this, a neural network was built to predict the positions of the defending players in short 9 second phases of play given the positions of the ball and attacking team. Different neural network architectures were used such as a Recurrent Neural Network and a Long Short Term Memory Network with a variety of activation functions, optimisers and other hyperparameters. The quality of the model's output was measured by minimising the squared distance between the predicted positions and the expected positions as well as the angle between the prediction and the centre of the goal. The best model successfully reproduced sequences from a whole game from 2016 reaching $\log(\text{loss}) = 8$ which corresponds to an average distance of 3 m between expected and predicted positions. However, the model heavily overfit to the training data and could not predict the testing sequences with a good enough accuracy. The testing loss had a minimum at $\log(\text{loss}) = 13.5$ which corresponds to an average distance of 11 m which is far from the optimum threshold. The model also failed to generalise to sequences over a whole season. These issues will be addressed in the future with methods such as further data processing, increasing model complexity and different data formats such as matrix maps. Despite the shortcomings of the model, it will act as a good foundation to make these improvements in the future.

References

[1] Kiersten M Ruff and Rohit V Pappu. Alphafold and implications for intrinsically disordered proteins. *Journal of Molecular Biology*, 433(20):167208, 2021.

[2] Tianyu Wu et al. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[3] Shantanu Ingle and Madhuri Phute. Tesla autopilot: semi autonomous driving, an uptick for future autonomy. *International Research Journal of Engineering and Technology*, 3(9):369–372, 2016.

[4] SMIEE Ariyo and Engr Olufemi. The future of ai in sports decision making. *The Cable (2022)*, 2022.

[5] Fotmob, "premier league possession statistics". https://www.fotmob.com/en-GB/leagues/47/stats/season/17664/teams/possession percentage team/premier-league-teams. Accessed: 04/01/2024.

[6] AD Dongare, RR Kharde, and Amit D Kachare. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(1):189–194, 2012.

[7] A Dureja and P Pahwa. Analysis of nonlinear activation functions for classification tasks using convolutional neural networks. In S Mishra, YR Sood, and A Tomar, editors, *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*, pages 1179–1190, Singapore, 2019. Springer Singapore.

[8] Karen Loaiza. *Deep learning for decision support in dermatology.* PhD thesis, 09 2020.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[10] 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.

[11] Jian Fu, JingChun Chu, Peng Guo, and Zhenyu Chen. Condition monitoring of wind turbine gearbox bearing based on deep learning model. *IEEE Access*, pages 1–10, 04 2019.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] [redacted]

[14] Ari Freedman. Convergence theorem for finite markov chains. *Proc. REU*, 2017.

Appendix

Model Hyperparameters

Hyperparameter	Value
Optimiser	ADAM
Learning Rate	0.01
Activation Function	Leaky ReLU
Hidden Layers	2
Hidden Dimensions	64
Train Test Split	80/20
Training Iterations	10^4
Drop Out Coefficient	0.2
Batch Size	1
Sequence Length	45 frames
Initialisation	He

Table 2: A table showing the hyperparameters used in the enhanced LSTM model.