Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Informatica per il Management

The Future of Formula 1 Racing: Neural Networks to Predict Tyre Strategy

Relatore:
Chiar.ma Prof.ssa
Elena Loli Piccolomini
Correlatore:
Dott.
Davide Evangelista

Presentata da:
Massimo Rondelli

Sessione III
Anno Accademico 2021/2022

To my family and
all the people I care about.

Estratto in lingua italiana

La Formula 1 è uno dei motorsport più popolari al mondo, con milioni di fan che seguono ogni gara. È uno sport altamente competitivo e tecnologicamente avanzato, con team che cercano costantemente modi per ottimizzare le loro prestazioni e ottenere un vantaggio competitivo. Uno dei fattori critici nella performance di un team è la scelta delle gomme durante una gara. Le gomme utilizzate nelle gare di Formula 1 sono fornite da Pirelli, il fornitore ufficiale di pneumatici per lo sport. Pirelli produce diverse tipologie di pneumatici, ognuno con caratteristiche uniche che li rendono più adatti a specifiche condizioni di pista e strategie di gara. Di solito sono disponibili tre diversi tipi di pneumatici per ogni gara, con diversi livelli di aderenza, durata e velocità. Le gomme più morbidi offrono maggior aderenza ma si consumano più velocemente, mentre quelle più dure durano più a lungo ma offrono meno aderenza. I team devono scegliere quali utilizzare durante la gara in base a una serie di fattori, tra cui la temperatura della pista, l'usura delle gomme e le condizioni meteorologiche. La strategia delle gomme è un aspetto essenziale della performance di un team, poiché fare le scelte giuste può fornire un vantaggio competitivo. I team devono bilanciare la necessità di tempi sul giro ottimali con la necessità di ridurre al minimo il numero di pit stop richiesti durante la gara. Ciò richiede un'analisi accurata dei dati, compresi i tassi di usura delle gomme e i tempi sul giro, per determinare il momento ottimale per cambiare le gomme e quale gomma utilizzare.

Negli ultimi anni, l'uso di modelli informatici e strumenti di analisi dei dati è diventato sempre più importante per ottimizzare la strategia delle gomme. I team utilizzano algoritmi sofisticati per analizzare grandi quantità di dati e prendere decisioni sui cambi di gomme durante la gara. L'uso di algoritmi di deep learning e di reti neurali nella previsione delle gomme è un campo emergente che ha il potenziale per fornire previsioni ancora più accurate e migliorare la performance della gara. In generale, la strategia delle gomme è un componente critico del successo in Formula 1 e la capacità di prevedere con precisione i cambi di gomme può fornire un significativo vantaggio competitivo. Utilizzando algoritmi di deep learning per prevedere la strategia delle gomme, i team possono

prendere decisioni più informate, ottimizzare la loro performance e aumentare le loro possibilità di successo in pista.

Lo scopo di questa tesi di laurea è quello di sviluppare e implementare reti neurali, in particolare LSTM e GRU, per prevedere la strategia delle gomme durante una gara di Formula 1. Il progetto mira a utilizzare i dati telemetrici provenienti dall'API "FastF1" e sviluppare un modello che possa prevedere con precisione quando e quali cambi di gomme sono richiesti durante una gara. L'obiettivo è quello di dimostrare il potenziale delle reti neurali nella previsione della strategia delle gomme e dimostrarne l'applicazione pratica in un ambiente ad alta performance come la Formula 1. I risultati del progetto possono fornire informazioni sull'efficacia di questi algoritmi in questo contesto e contribuire allo sviluppo di modelli di previsione delle gomme più avanzati in futuro.

Contents

Estratto in lingua italiana **1**

1 Tyre Strategy in Formula 1 **5**

2 Introduction to Deep Learning **7**
2.1 Logical Computations with Neurons 7
2.2 The Perceptron 8
2.2.1 How to train a perceptron 9
2.3 Deep Neural Network 10
2.3.1 How does backpropagation work? 10
2.3.2 The Vanishing/Exploding Gradients Problems 12
2.3.3 Activation functions 13
2.3.4 Hidden Layers 13
2.3.5 Learning Rate and Optimizer 14
2.3.6 Batch Normalization 16
2.3.7 Dropout 17
2.4 Recurrent Neural Networks 18
2.4.1 How to train RNNs 20
2.4.2 Long short-term memory (LSTM) 21
2.4.3 Gated Recurrent Unit (GRU) 23

3 Experimental Setup **25**
3.1 FastF1 library 25
3.2 Implementation 26
3.2.1 First Experiment 27
3.2.2 Second Experiment 32
3.2.3 Neural network models 36

4 Numerical Results **43**
4.1 LSTM vs GRU 43
4.2 GRU vs MLP 46
4.3 Blind Classifier 47

5 Conclusion **53**

Chapter 1

Tyre Strategy in Formula 1

Formula 1 is one of the most popular motorsports in the world, with millions of fans following every race. It is a highly competitive and technologically advanced sport, with teams constantly seeking ways to optimize their performance and gain a competitive edge. One of the critical factors in a team's performance is the choice of tyre compounds during a race. The tyres used in Formula 1 races are provided by Pirelli, the official tyre supplier for the sport. Pirelli produces several different tyre compounds, each with unique characteristics that make them more suitable for specific track conditions and racing strategies. There are typically three different tyre compounds available for each race, with varying levels of grip, durability, and speed. The softer compounds provide more grip but wear out more quickly, while the harder compounds last longer but offer less grip. Teams must choose which compounds to use during the race based on a variety of factors, including track temperature, tyre wear, and weather conditions. Tyre strategy is an essential aspect of a team's performance, as making the right choices can provide a competitive advantage. Teams must balance the need for optimal lap times with the need to minimize the number of pit stops required during the race. This requires careful analysis of data, including tyre wear rates and lap times, to determine the optimal time to change tyres and which compounds to use.

In recent years, the use of computer modeling and data analysis tools has become increasingly important in optimizing tyre strategy. Teams use sophisticated algorithms to analyze large amounts of data and make informed decisions on tyre changes during the race. The use of deep learning and neural network algorithms in tyre prediction is an emerging field that has the potential to provide even more accurate predictions and improve race performance.

Overall, tyre strategy is a critical component of success in Formula 1, and the ability to accurately predict tyre changes and compounds can provide a significant competitive

advantage. By using deep learning and neural network algorithms to predict tyre strategy, teams can make more informed decisions, optimize their performance, and increase their chances of success on the track.

The aim of this bachelor's thesis is to develop and implement deep learning and neural network algorithms, specifically LSTM, and GRU, to predict tyre strategy during a Formula 1 race. The project aims to utilize telemetry data sourced from the "fastf1" API and develop a model that can accurately predict when and which tyre compound changes are required during a race. The objective is to showcase the potential of neural networks in predicting tyre strategy and demonstrate its practical application in a high-performance environment such as Formula 1. The results of the project can provide insights into the effectiveness of these algorithms in this context and contribute to the development of more advanced tyre prediction models in the future.

Chapter 2

Introduction to Deep Learning

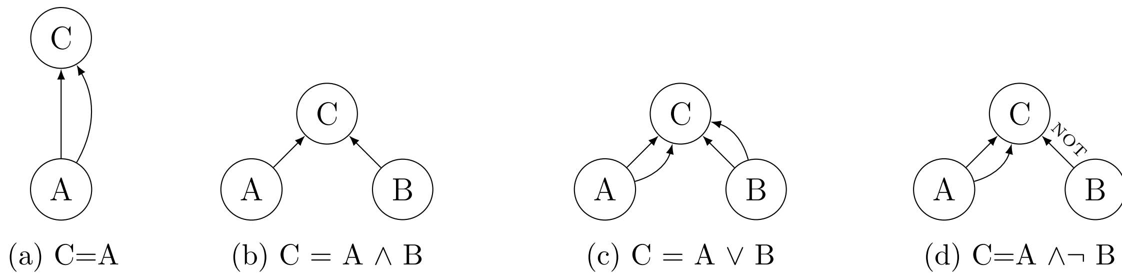
It seems logical to look at the brain's architecture for inspiration to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANN_s). Machine learning models are inspired by the networks of biological neurons in our brains. ANN_s are at the very core of deep learning. The first part of this chapter introduces artificial neural networks, starting with a quick visit to the very first ANN architectures and leading up to multilayer perceptrons, which are very used today.

2.1 Logical Computations with Neurons

The neurophysiologist Warren McCulloch and the mathematician Walter Pitts proposed, in 1943, a very simple model of the biological neuron, which became known as *artificial neuron*. It has one or more binary inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active.

Let's see what these networks do:

- The network 2.1a says: if neuron A is activated, then neuron C gets activated as well since it receives two input signals from neuron A. on the contrary if A is off, C is off as well.
- The network 2.1b performs a logical *AND*. Neuron C is activated only when both neurons A and B are activated.
- The network 2.1c performs a logical *OR*. Neuron C gets activated if either neuron A or B is activated. Or both.



(a) C=A (b) C = A ∧ B (c) C = A ∨ B (d) C=A ∧¬ B

Figure 2.1: Artificial Neural Networks (ANNs) Performing Logical Computations - ANNs use multiple layers of neurons to perform logical computations.

- The network 2.1d performs a complex logical proposition. Neuron C is activated only if neuron A is active and neuron B is off. They must follow this schema, otherwise, neuron C won't be activated.

2.2 The Perceptron

The perceptron is one of the simplest ANN architectures. It is based on an artificial neuron called *threshold logic unit* (TLU), Figure 2.2. The inputs and outputs are numbers, instead of binary on/off values, and each input connection is associated with a weight. The TLU computes a linear function of its inputs:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b = w^T \cdot x + b$$

After that, it applies a step function to the result:

$$h_w(x) = f(z)$$

The model parameters are the input weights $\mathbf{w}$ and the bias term b. A single TLU, can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. It is called *fully connected layer*. The inputs constitute the *input layer* and the final output is called *output layer*. It's possible to compute the outputs of a layer for more instance at once, just by applying the following formula:

$$h_{W,b}(X) = \phi(XW + b)$$

The parameters are:

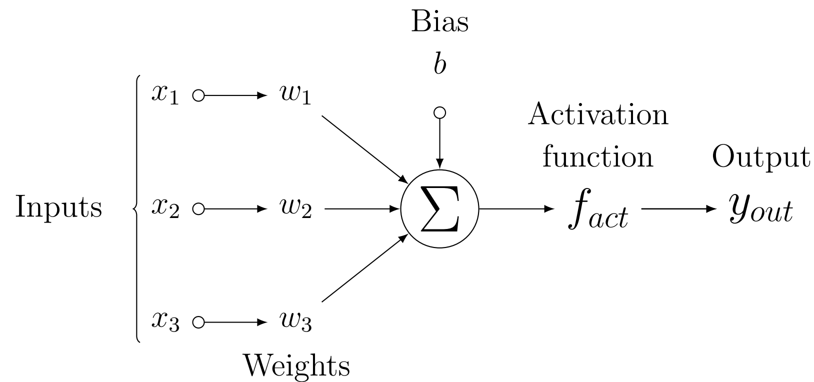


Figure 2.2: The Threshold Logic Unit (TLU) - A fundamental building block of artificial neural networks, implementing a linear decision boundary to classify input data.

- X represents the matrix of input features.
- The weight matrix W contains all the connection weights.
- The bias vector b contains all the bias terms (one per neuron).
- The function ϕ is called the *activation function*. When the artificial neuron is a TLU, it is called a step function (the activation function will be discussed in the following chapter).

2.2.1 How to train a perceptron

Donald Hebb, considered the "father of neuropsychology", in his 1949 book *The Organization of Behavior*, suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. The connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule. Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction. The perceptron learning rule reinforces connections that help reduce the error. The rule is shown in the following equation:

$$w_{i,j}^{\text{next step}} = w_{i,j} + \eta(y_i - \hat{y}_j) \cdot x_i$$

where:

- $w_{i,j}$ is the connection weight between the ith input and the jth neuron.

- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate.

This perceptron training algorithm was proposed by Frank Rosenblatt who he got his inspiration from Hebb's rule. This algorithm is also called *perceptron convergence theorem.*

2.3 Deep Neural Network

A Multilayer Perceptron (MLP), Figure 2.3, is composed of one input layer, one or more layers of TLUs *hidden layers*, and one final layer of TLUs called the *output layer*. When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN). For many years researchers struggled to find a way to train MLPs. In the 1960s, some researchers discussed the possibility of using gradient descent to train neural networks but just in 1970, a researcher named Seppo Linnainmaa introduced a technique to compute all the gradients automatically and efficiently. This algorithm is called *reverse-mode automatic differentiation*. In two passed through the network (one forward, one backward), it can compute the gradients of the neural network's error about every single model parameter. It can find out how each connection weight and each bias should be tweaked to reduce the neural network's error. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode automatic differentiation is called *backpropagation*.

Backpropagation can be applied to all sorts of computational graphs, not just neural networks. In 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper [8] analyzing how backpropagation allowed neural networks to learn useful internal representations. Today, it is the most popular way to train neural nets.

2.3.1 How does backpropagation work?

It handles one mini-batch at a time, and it goes through the full training set multiple times. Each pass is called an *epoch*. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first

hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*.

The algorithm measures the network's output error. To calculate the error, it uses a loss function that compares the desired output and the actual output of the network and returns some measure of the error. Then, it computes how much each output bias and each connection to the output layer contributed to the error. The algorithm then measures how much of these error contribution came from each connection in the layer below, working backward until it reaches the input layer. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients just computed. It is important to initialize all the hidden layer's connection weights randomly, or else training will fail. If you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly same way, so they will remain identical. If instead, you randomly initialize the weights, you break the symmetry and allow backpropagation to train the neurons. Summing up, we can say that this technique makes predictions for a mini-batch, the forward pass, measures the error, then goes through each layer in reverse to

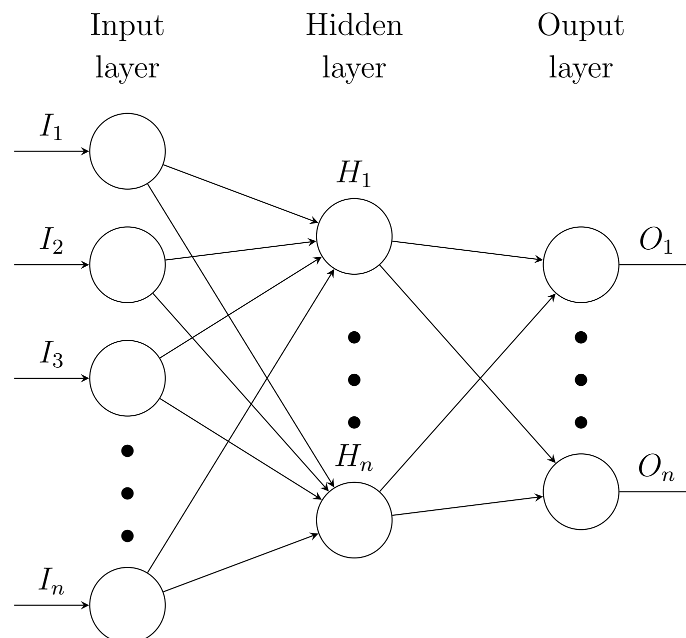


Figure 2.3: Architecture of a Multilayer Perceptron - A type of feedforward neural network that consists of multiple layers of interconnected neurons, with each neuron in one layer connected to every neuron in the next layer.

measure the error contribution from each parameter, backward pass, and finally tweaks the weights and biases of the connections to reduce the error, gradient descent step.

2.3.2 The Vanishing/Exploding Gradients Problems

The second stage of the backpropagation algorithm propagates the error gradient while moving from the output layer to the input layer. The algorithm uses these gradients to update each parameter with a gradient descent step after computing the gradient of the cost function for each network parameter. Unfortunately, when the algorithm descends to the lower layers, the gradient frequently gets smaller and smaller. As a result, training never converges to a good solution and the gradient descent update essentially leaves the connection weights of the lower layers unchanged. The vanishing gradient problem is what's happening here. The inverse can also happen in some situations, causing the gradients to get larger until the layers receive massive weight updates and the algorithm diverges. This is the exploding gradients problem, which recurrent neural networks encounter most frequently. Deep neural networks more typically experience unstable gradients.

A paper [3] by Xavier Glorot and Yoshua Bengio published in 2010 identified a few suspects, including the popular weight initialization technique and the combination of the sigmoid activation function. They demonstrated that the variation of each layer's outputs is significantly higher than the variance of its inputs when using this activation function and initialization technique. The activation function reaches saturation at the top layers as the network advances, with the variance increasing after each layer. The sigmoid function's mean is 0.5 rather than 0, which makes this saturation worse. When inputs are high (positive or negative), as can be seen by looking at the sigmoid activation function, Figure 2.4, the function saturates at 0 or 1, with a derivative that is very close to 0. As a result, when backpropagation begins, no gradient is left for it to propagate back through the network.

Using weight initialization, where the weights are small random values, can help to prevent the exploding gradient problem. As presented in the paper, one popular initialization technique is called *"Xavier"* or *"Glorot"*, which adjusts the scale of the weights based on the number of input and output neurons in the layer. Another way to solve these problems is by using non-linear activation functions. Using non-linear activation functions such as ReLU, leaky ReLU, or Maxout instead of sigmoid or tanh can help mitigate the vanishing gradient problem (we are going to see activation functions in the following section). The last two techniques we can use to avoid this problem are *batch normalization* and *regularization*, which will be shown in the following sections.

2.3.3 Activation functions

Rumelhart revised the structure of MLP to make backprop function properly: they replaced out the step function for the logistic function, better known as the *sigmoid function*. This was crucial because there is no gradient to work with in the step function because the gradient descent cannot move on a flat surface, allowing it to advance somewhat at each step. The step function only contains flat segments. In actuality, aside from the sigmoid function, the backpropagation algorithm performs well with a wide variety of alternative activation functions. Here are two other popular choices. In the Figure 2.4, it's possible to see how they work.

- *The hyperbolic tangent function:* $tanh(z) = 2\sigma(2z) - 1$

  This activation function is S-shaped, continuous, and differentiable just like the sigmoid function, however its output value spans from -1 to 1, as opposed to 0 to 1, in the case of the sigmoid function. Because of this range, the output of each layer is more or less centered at zero at the start of training.

- *The rectified linear unit function:* $ReLU(z) = max(0, z)$

  Although continuous, the ReLU function is unfortunately not differentiable at $z = 0$ and its derivative is 0 for $z < 0$. But since it performs so well in practice and offers the benefit of being quick to compute, it has taken over as the standard. Since biological neurons appear to implement an activation function that is roughly sigmoid (S-shaped), researchers have focused on sigmoid functions for a very long time. ReLU, however, really performs better in ANNs in general. The biological analogy may have been misleading in this instance.

Why do we need activation functions? All you get when you combine many linear transformations is another linear transformation. Therefore, if there is no nonlinearity between the levels, even a deep stack of layers is equivalent to one layer, making it impossible to handle extremely complicated issues.

2.3.4 Hidden Layers

Many issues can be solved by starting with a single hidden layer and producing acceptable results. Even the most complex functions can theoretically be modeled by an MLP with just one hidden layer. Deep networks, however, outperform shallow ones in terms of *parameter efficiency* for complicated situations. Deep neural networks benefit from the fact that real-world data is frequently hierarchically structured as follows: The output

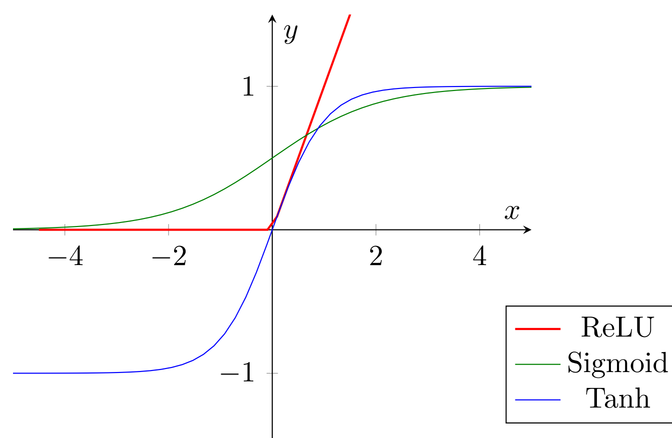


Figure 2.4: Activation Functions (ReLU, Sigmoid, Tanh) - Activation functions determine the output of a neuron in a neural network, and ReLU, Sigmoid, and Tanh are some commonly used activation functions that enable non-linear transformations of input data.

layer and the highest hidden layers combine these intermediate structures to model high-level structures. Lower hidden layers model low-level structures, intermediate hidden layers combine these low-level structures to model intermediate-level structures, and the highest hidden layers model high-level structures like faces.

In conclusion, the neural network will function properly in many situations if you start with just one or two hidden layers. Increase the number of hidden layers for more complicated problems until the training set starts to become overfit. Large picture classification or speech recognition are two examples of extremely hard jobs that generally demand networks with dozens of layers and a huge amount of training data.

2.3.5 Learning Rate and Optimizer

There are more hyperparameters in an MLP than the number of neurons and hidden layers that can be changed. Some of the most important are listed below:

Learning rate

The learning rate is the most important hyperparameter. The ideal learning rate is often equal to half the maximum learning rate. Training the model for a few hundred iterations with a very low learning rate, like 10^{-5}, and progressively raising it to a very high number, like 10, is one method of determining a good learning rate. If you plot the loss as a function of the learning rate, you should notice that it first decreases. However,

after a while, the learning rate will become excessive, causing the loss to quickly increase again. The optimal learning rate will be slightly lower than the point at which the loss begins to increase.

Optimizer
Equally important is selecting a better optimizer than just an odd mini-batch gradient descent. There are several optimizers that you can use to speed up the training but we are going to see just one, *Adam*. Adam [7], which stands for *adaptive moment estimation*, combines the concepts of momentum optimization and RMSProp: it tracks an exponentially decaying average of previous gradients, just like momentum optimization, and just like RMSProp, it tracks an exponentially decaying average of previously squared gradients. These are estimates of the gradients' mean and variance. The algorithm's name comes from the fact that the mean is sometimes referred to as the first instant and the variance as the second. In other words, we can say the algorithm keeps track of two moving averages: the mean and the variance of the gradients; these moving averages are updated at each training step. This allows the algorithm to adapt to changes in the distribution of the gradients, which can be very beneficial for training deep neural networks:

$$t = t + 1 \tag{2.1}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\theta J(\theta) \tag{2.2}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_\theta J(\theta))^2 \tag{2.3}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.5}$$

$$\theta = \theta - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.6}$$

where θ represents the parameters of the model, $\nabla_\theta J(\theta)$ is the gradient of the cost function J with respect to the parameters θ, β_1 and β_2 are the decay rates of the moving averages, α is the learning rate, and ϵis a small constant added to the denominator to prevent division by zero.

At each time step t, the mean m_t and the variance v_t of the gradients are updated using the equations (2.2) and (2.3). Here, m_{t-1} and v_{t-1} represent the moving averages

of the gradients and their squares from the previous time step. β_1 and $beta_2$ are hyperparameters that control the decay rate of the moving averages. They determine the amount of weight given to the past values of the gradients and their squares and the amount of weight given to the current gradient and its square. After updating m_t and v_t, the moving averages are corrected for bias using the equations (2.4) and (2.5). The bias correction ensures that the moving averages are a better estimate of the true mean and variance of the gradients, especially at the start of the optimization process when t is small. Finally, the parameters θ are updated using the equation (2.6). Here, α is the learning rate, which determines the step size at which the parameters are updated. The term $\frac{\hat{m}_t}{\sqrt{\hat{v}_t+\epsilon}}$ is an approximation of the gradient of the cost function J with respect to the parameters θ. The addition of ϵ in the denominator is used to prevent division by zero.

This update rule balances the magnitude of the steps taken in the direction of the gradient with the magnitude of the gradient itself, allowing the optimization algorithm to make large steps in the directions with a high gradient and small steps in the directions with a low gradient. This makes the optimization process more efficient and helps the algorithm avoid getting stuck in local minima. There are three variants of Adam: AdaMax, Nadam, and Adam W.

2.3.6 Batch Normalization

The danger of the vanishing/exploding gradients problems can be considerably reduced at the beginning of training by employing Glorot initialization in conjunction with ReLU, but this does not ensure that they won't reappear later on. In a 2015 paper [6], Sergey Ioffe and Christopher Szegedy suggested a method to solve these issues called *batch normalization* (BN). The procedure includes inserting an operation into the model just before or after each hidden layer's activation function. The technique helps the model discover the ideal mean and scale for each input layer. In many circumstances, standardizing your training set is unnecessary if you include a BN layer as the initial layer of your neural network.

Let's say that we have a mini-batch of N examples, and the activations for a particular layer for one example is given by $X = [x_1, x_2, ..., x_d]$, where d is the number of neurons in the layer. The batch normalization algorithm consists of the following steps:

1. Calculate the mean and variance of the activations for the mini-batch:

$$\text{mean} = \frac{1}{N} \sum_{i=1}^{N} X_i \tag{2.7}$$

$$\text{variance} = \frac{1}{N} \sum_{i=1}^{N} (X_i - \text{mean})^2 \tag{2.8}$$

2. Normalize the activations:

$$\hat{X} = \frac{X - \text{mean}}{\sqrt{\text{variance} + \epsilon}} \tag{2.9}$$

   where ϵ is a small constant added for numerical stability.

3. Scale and shift the normalized activations:

$$X_{\text{bn}} = \gamma \cdot \hat{X} + \beta \tag{2.10}$$

   where γ and β are learnable parameters.

4. Use the normalized and scaled activations as inputs to the next layer in the network.

It is important to note that during training, the mean and variance of the activations are calculated on each mini-batch. During inference, the mean and variance are estimated using a running average that is updated during training. This allows the batch normalization layer to normalize the activations in a way that is consistent with the training data. In summary, the batch normalization algorithm normalizes the activations of a layer in a deep neural network by subtracting the mean and dividing by the standard deviation, scaling and shifting the result using learnable parameters, and using the normalized activations as inputs to the next layer in the network.

In conclusion, batch normalization is a widely used and effective technique for improving the training and performance of deep neural networks. It can help speed up training, prevent overfitting, and make the training process more robust to changes in the scale of the inputs and weights.

2.3.7 Dropout

Dropout, Figure 2.5, is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper [4] by Geoffry Hinton et al. in 2012. It is a relatively straightforward algorithm: at each training step, each neuron (including input neurons but never output neurons) has a probability p of being temporarily "dropped

out," which means it will be completely ignored during this training phase but may be active during the next. This means that their activations are set to zero, and their incoming and outgoing connections are ignored. This has the effect of reducing the number of parameters in the network and making it more difficult for the network to memorize the training data. The dropout rate, also known as the hyperparameter p, is normally set between 10% and 50%; in recurrent neural networks, it is more likely to be between 20% and 30%.

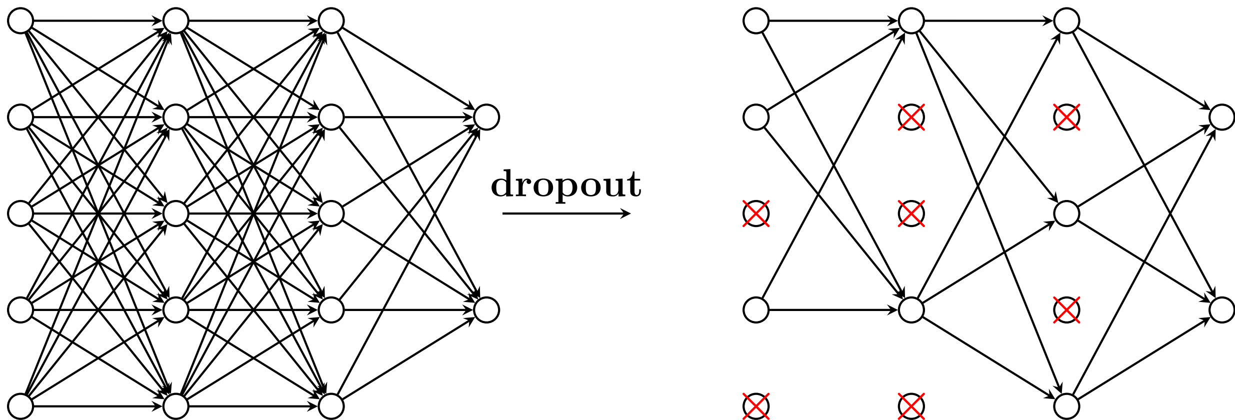


Figure 2.5: Dropout Scheme - A regularization technique used in neural networks to prevent overfitting by randomly dropping out some of the neurons in the network during training.

For example, if the dropout rate is 0.5, then during each training iteration, on average, half of the units in the network will be dropped out. This results in a different, randomly perturbed network architecture at each training iteration, which helps prevent overfitting. The dropout rate can be tuned through experiments to find the optimal value for a given problem and network architecture. In general, a dropout rate of 0.5 is a good starting point, but the optimal value will depend on the specifics of the problem and the network architecture.

2.4 Recurrent Neural Networks

A class of nets called recurrent neural networks (RNNs) is capable of foreseeing the future. RNNs are capable of analyzing a variety of time series data, like the number of daily visitors to your website, the local hourly temperature, and more. An RNN can forecast the future using its knowledge of past patterns in the data, presuming of course

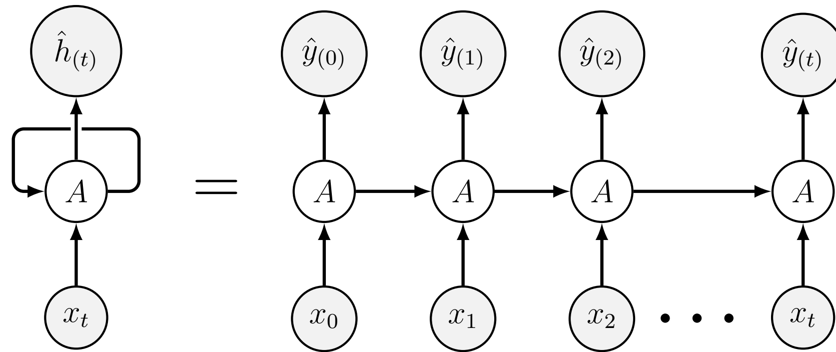


Figure 2.6: A Recurrent Neuron (left) Unrolled through Time (right) - Recurrent neural networks (RNNs) are used to model sequences of data, with each neuron in the network receiving input not only from the current time step but also from the previous time step.

that those patterns will continue to exist. Similar in appearance to a feedforward neural network, a recurrent neural network also includes connections pointing backward.

Let's discuss the simplest possible RNN, which consists of a single neuron taking inputs, producing output, and sending that output back to the neuron that received it, Figure 2.6 (left). This recurrent neuron receives its own output from the previous time step $\hat{y}_{(t-1)}$ and the inputs $x_{(t)}$ at each time step t. At the first time step, the output is set at 0, since there is no output at the previous time step. This little network can be shown in relation to the time axis, Figure 2.6 (right). "Unrolling the network through time" is what is meant by this. Each neuron receives the output vector from the previous time step, $\hat{y}_{(t-1)}$, as well as the input vector $x_{(t)}$, at each time step t. As you can see, now inputs and outputs are vectors. One set of weights is for the inputs $x_{(t)}$, and the other is for the outputs of the previous time step $\hat{y}_{(t-1)}$ for each recurrent neuron. These weight vectors will be abbreviated w_x and $w_{\hat{y}}$. We can organize all the weight vectors into two weight matrices, W_x and $W_{\hat{y}}$, if we think about the entire recurrent layer rather than just one recurrent neuron. The output vector of the entire recurrent layer can then be calculated in a similar way to what one might anticipate.

$$\hat{y}_{(t)} = \sigma\left(W_x^T x_{(t)} + w_{\hat{y}}^T \hat{y}_{(t-1)} + b\right) \tag{2.11}$$

Like feedforward neural networks, by putting all the inputs at time step t into an input matrix X, we can compute the output of a recurrent layer in one single step for an entire mini-batch.

$$\begin{aligned}\hat{Y}_{(t)} &= \sigma\left(X_{(t)}W_x + \hat{Y}_{(t-1)}W_{\hat{y}} + b\right)\\ &= \sigma\left(\begin{bmatrix}X_{(t)} & \hat{Y}_{(t-1)}\end{bmatrix} W + b\right) \text{ with } W = \begin{bmatrix}W_x\\ W_{\hat{y}}\end{bmatrix}\end{aligned} \tag{2.12}$$

In above equation, (2.12), we can see:

- $\hat{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's output at time step t for each instance in the mini batch.
- $X_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances.
- W_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $W_{\hat{y}}$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- b is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices W_x and $W_{\hat{y}}$ are concatenated vertically into a single weight matrix W.
- The notation $\begin{bmatrix}X_{(t)} & \hat{Y}_{(t-1)}\end{bmatrix}$ represents the horizontal concatenation of the matrices $X_{(t)}$ and $\hat{Y}_{(t-1)}$.

Notice that $\hat{Y}_{(t)}$ is a function of $X_{(t)}$ and $\hat{Y}_{(t-1)}$, which is a function of $X_{(t-1)}$ and $\hat{Y}_{(t-2)}$, which is a function of $X_{(t-2)}$ and $\hat{Y}_{(t-3)}$, and so on. This makes $\hat{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is $X_{(0)}, X_{(1)}, X_{(2)}, ..., X_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are assumed to be all zeros.

2.4.1 How to train RNNs

You could say that a recurrent neuron has a form of memory because its output at a given time step t is a function of all its inputs from earlier time steps. A *memory cell* is a component of a neural network that keeps a certain state over successive time steps. The state of a cell at time step t, represented by the symbol $h_{(t)}$, is a function of some inputs at that time step and its state at the previous time step. So, we can say $h_{(t)} = f\left(x_{(t)}, h_{(t-1)}\right)$. The previous state and the current inputs are functions of the output at time step t, indicated as $\hat{y}_{(t)}$.

An RNN can accept a series of inputs and generate different sequences:

- *Sequence-to-sequence network:* it takes a sequence of inputs and produces a sequence of outputs at each time step t.
- *Sequence-to-vector network:* it takes a sequence of inputs, and you can consider only some outputs. For example, if you have 5 inputs, you might want only the last output, so you can ignore all the previous outputs.
- *Vector-to-sequence network:* The input sequence is a vector that you pass into the network at each time step and let it output a sequence.
- *Encoder-decoder network:* This network is mostly used for translations. You pass a sentence in one language, and the output will be translated in another language.

The idea is to unroll an RNN over time before using traditional backpropagation to train it. The term *backpropagation over time* (BPTT) refers to this technique. The network is initially passed forward after it has been unrolled. After that, a loss function is used to evaluate the output sequence.

$$L(Y_{(0)}, Y_{(1)}, ..., Y_{(T)}; \hat{Y}_{(0)}, \hat{Y}_{(1)}, ..., \hat{Y}_{(T)}) \tag{2.13}$$

where $Y_{(i)}$ is the ith output, $\hat{Y}_{(i)}$ is the ith prediction and T is the max time step. For example, if we think about *sequence-to-vector network*, we want to compute only just the last two outputs of the network, ignoring the first three outputs. It means that the loss function isn't computed on all outputs, but just on the last two.

The unrolled network then propagates the gradients of that loss function backward. The gradients only pass through the outputs $\hat{Y}_{(3)}$ and $\hat{Y}_{(4)}$, since in the example the outputs $\hat{Y}_{(0)}$, $\hat{Y}_{(1)}$ and $\hat{Y}_{(2)}$ are not used to calculate the loss. Thusly, because W and b are identical parameters at every time step, their gradients will be changed numerous times during backprop. The parameters can be updated using a gradient descent step using BPTT when the backward phase is finished, and all the gradients have been computed. This is how RNN training is made.

2.4.2 Long short-term memory (LSTM)

In 1997, Seep Hochreiter and Jürgen Schmidhuber proposed the "Long Short-Term Memory" (LSTM) cell, which was progressively improved over time by other researchers [5]. If the LSTM cell is viewed as a black box, it can be used in a similar way to that of a

basic cell but will perform much better. Training will converge more quickly and find longer-term patterns in the data.

How do LSTM cells work? Figure 2.7 represents its architecture. The LSTM cell appears just like a standard cell from the outside, except for the fact that its state has been divided into two vectors, $h_{(t)}$ and $c_{(t)}$. The short-term state is represented by $h_{(t)}$, and the long-term state is represented by $c_{(t)}$.

The main concept is that the network can learn what to read from it, and what to discard or store in the long-term state. You can see that as the long-term state $c_{(t-1)}$ moves from left to right throughout the network, it first uses a forget gate to delete some memories before adding some new ones using an addition operation that includes memories that were chosen by an input gate. Without any further change, the result $c_{(t-1)}$ is sent out directly. Several memories are added, and some are deleted at each time step. The long-term state is copied and then passed via the tanh function following the addition operation. The output gate then filters the outcome. This produces the short-term state $h_{(t)}$.

Now let's see how gates perform. First, four separate fully connected layers take the current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$. Each one has a specific function:

- The layer that outputs $g_{(t)}$ is the primary layer. Its regular functions include

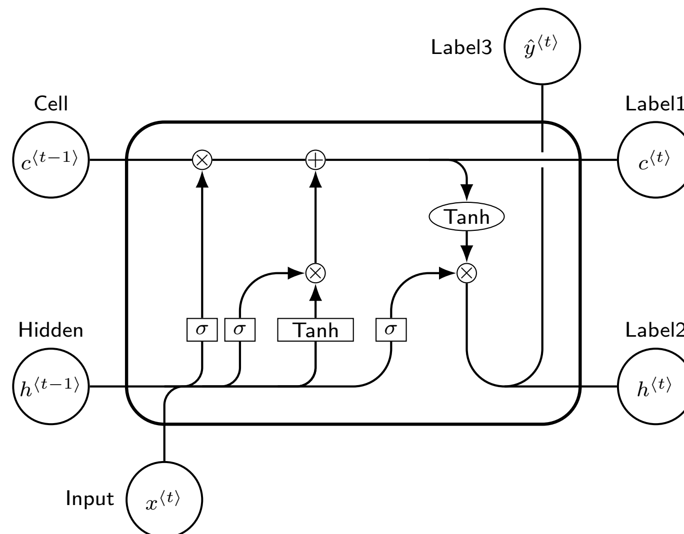


Figure 2.7: An LSTM Cell - A type of recurrent neural network cell that can selectively remember or forget information from previous time steps, making it particularly useful for processing sequential data such as text and speech.

processing the inputs for the present $x_{(t)}$ and the past $h_{(t-1)}$ states. The output of this layer is not sent directly outside. Instead, its most fundamental parts are stored in the long-term state. The rest is dropped.

- Gate controllers are the three additional layers. The outputs are in the sigmoid activation function range, which is 0 to 1. The outputs from the gate controllers are fed into element-wise multiplication processes; if the output is a 0, the gate is closed; if a 1, the gate is opened. Particularly:
  - The *forget gate* $f_{(t)}$: determines which elements of the long-term state should be removed.
  - The *input gate* $i_{(t)}$: determines which $g_{(t)}$ components go into the long-term state.
  - The *output gate* $o_{(t)}$: determines which elements of the long-term state should be read and output at this time step, both to $h_{(t)}$ and $y_{(t)}$,

In conclusion, we can say that an LSTM cell can understand how to identify important input, *"input gate"* role, and it can store in a long-term state, preserve and use it whenever it wants, *"forget gate"* role. There are more variants of the LSTM cell. Let's see now, the most used and important: the *GRU* cell.

2.4.3 Gated Recurrent Unit (GRU)

In a 2014 paper, Kyunghyun Cho et al. made the suggestion for the Gated Recurrent Unit (GRU) cell [1]. The GRU cell, which is an LSTM cell simplified, seems to work just as well. The main changes are as follows:

- A single vector $h_{(t)}$ is created by combining the two state vectors.
- Both the input gate and the forget gate are managed by a single gate controller $z_{(t)}$. The input gate is closed $(1 - 1 = 0)$ and the forget gate is open $(= 1)$ if the gate controller sends a 1. The opposite occurs if the output is a 0. To put it another way, whenever a memory needs to be saved, the area where it will be stored must first be deleted.
- The entire state vector is output at each time step; there is no output gate. The main layer $g_{(t)}$ will only see certain portions of the prior state, thanks to a new gate controller $r_{(t)}$.

One of the key elements in the success of RNNs is the use of LSTM and GRU cells.

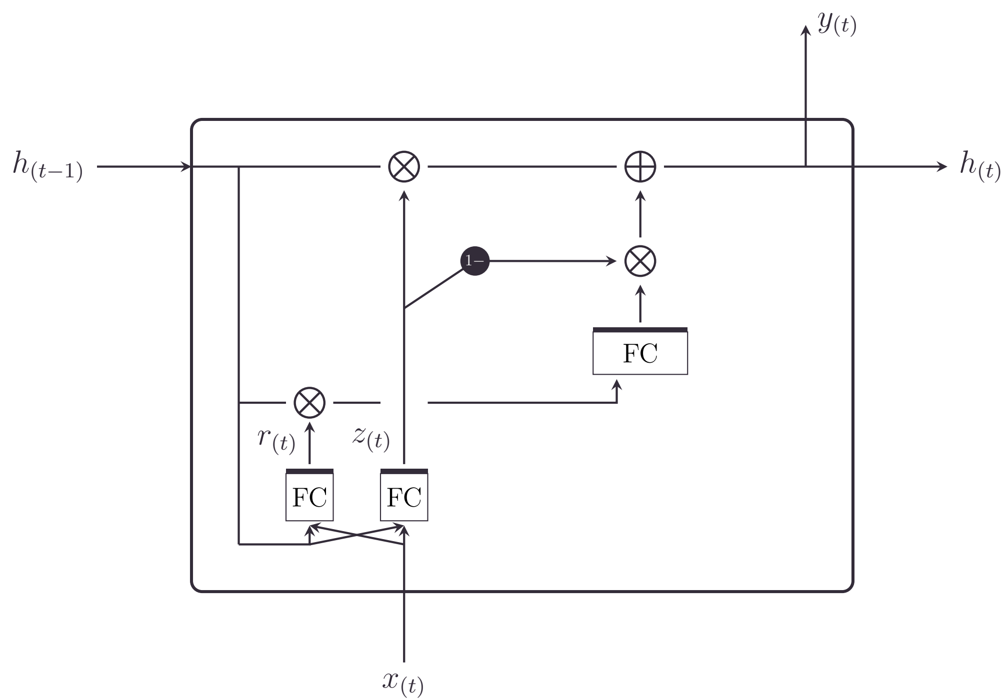


Figure 2.8: An GRU Cell - A type of recurrent neural network cell that uses gating mechanisms to control the flow of information through the cell, allowing it to selectively update or retain information from previous time steps.

Chapter 3

Experimental Setup

3.1 FastF1 library

FastF1 is an open-source Python library that provides a way to access telemetry data from Formula 1 races. The library allows users to access a variety of data points, such as lap times, tyre wear, fuel usage, and more, from the cars on the track during a race. These data are collected by the teams themselves and made available to the public after each race. For this project, I utilized the FastF1 library to obtain telemetry data for a variety of races throughout the Formula 1 season. I used the library to extract data on tyre compounds and pit stops made by each driver during the race, as well as other data points such as lap times, and weather conditions. This data was then preprocessed and cleaned before being used to train and evaluate our neural network models. One of the advantages of using the FastF1 library is that it provides a standardized format for accessing telemetry data from all races in the season. This allows for easy comparison of data across different races and helps to ensure that the data used in our models are consistent and reliable. Another benefit of using the FastF1 library is that it provides a high level of granularity in the data it provides. This allowed us to obtain detailed information on the timing and duration of pit stops, as well as the tyre compounds used by each driver during the race. This level of detail is important for accurately predicting tyre strategy and ensuring that our models can provide actionable insights for teams to use during the race. Overall, the FastF1 library provided a robust and reliable source of telemetry data for our project, allowing us to develop and test our neural network models with confidence.

First of all, to get access to FastF1 is necessary to install it.

```
pip install fastf1
```

It is required to load the grand prix session in order to get the information from the API. Three factors are required to do that: the *grand prix year*, the *location*, and the *session type*, which can be:

- Free Practice 1 (FP1).
- Free Practice 2 (FP2).
- Free Practice 3 (FP3).
- Qualifying (Q).
- Race (R).

```
import fastf1 as ff1

session = ff1.get_session(2022, 'Imola', 'R')
session.load()
```

After loading the session, it is possible to get all the data available in the library, such as lap time, compound, weather conditions, and so on.

3.2 Implementation

Formula 1 is a sport where every second counts. One critical aspect of a successful race is choosing the right type of tyre for the track conditions. In this thesis, I conducted two experiments. In the first one, I trained a neural network to predict the compound used by drivers during a lap. It is actually a useless prediction since the target is given in the last column of the dataset. But it has been done for a couple of reasons: learn how to create a 3D dataset and learn how to create and train a neural network. In the second experiment, instead, I used a similar approach to predict the best tyre for each lap during the race. Let's see now, in the details, the experiments.

3.2.1 First Experiment

As I said before, the first experiment has been done only for learning reasons. The first step to using a neural network, and so making predictions, is dataset creation. The first step is to collect the data from the library and store them in a `pandas` dataframe. To do that, two-parameter are necessary, the `driver` and the `session`. These inputs are mandatory to make the API request, otherwise, data won't be taken. The following code is how dataframe X is made:

```
1  # Function creates a df for a specific driver and session.
2  def get_data(driver, session):
3      session_driver = session.laps.pick_driver(driver)
4  
5      driver_lap_number = session_driver['LapNumber'] # Driver's lap
6      driver_sector1_time = (session_driver['Sector1Time'] / np.
         timedelta64(1, 's')).astype(float) # Sector 1 recorded time
7      driver_sector2_time = (session_driver['Sector2Time'] / np.
         timedelta64(1, 's')).astype(float) # Sector 2 recorded time
8      driver_sector3_time = (session_driver['Sector3Time'] / np.
         timedelta64(1, 's')).astype(float) # Sector 3 recorded time
9      driver_lap_time = session_driver['LapTime'].apply(
         timedelta_to_seconds)
10 
11     weather_rainfall = session.laps.get_weather_data()['Rainfall'] #
         Shows if there is rainfall
12     weather_rainfall = np.where(weather_rainfall == True, 1, 0)
13     weather_track_temperature = session.laps.get_weather_data()['
         TrackTemp'] # Track temperature [Celsius]
14 
15     driver_list = [driver] * len(driver_lap_number)
16     grand_prix_list = [session.event['Location']] * len(
         driver_lap_number)
17 
18     compound = session_driver['Compound']
19 
20     list_of_tuples = list(zip(driver_list, grand_prix_list,
         driver_lap_number, driver_sector1_time, driver_sector2_time,
         driver_sector3_time, driver_lap_time, weather_rainfall,
         weather_track_temperature, compound))
21     df = pd.DataFrame(list_of_tuples, columns = ['Driver', 'Race', '
         Lap', 'Sector 1 Time', 'Sector 2 Time', 'Sector 3 Time', 'Lap
         Time', 'Rainfall', 'Track Temp', 'Compound'])
22 
```

```
23     return df
```

We first selected a particular session and driver to focus on. We then needed to choose a *temporal resolution*, which we represented as T. T represents the lap number of the race and determines the granularity of the dataset. Once we had chosen T, we needed to decide what information we wanted to collect for each time $t \in [0, T]$. We chose a set of n different types of information. We then collected this information for each time $t \in [0, T]$ of the race, and stored it in X. Each row of X represents a single lap of the race, and contains n columns, one for each type of information we collected, such as sectors time, lap time, track temperature, rainfall and compound. As you can see, the last information in the dataset is the *compound*. We know for each lap which is the driver's tyre. Like this, the dataframe has data for each lap in the given session.

This is the first step to creating the dataset that neural networks require. Since we are predicting tyre changes over time, we need to represent the input data in a 3D format. Specifically, we need to include a temporal dimension to the data, since the tyre changes are dependent on the lap number of the race. By representing the data as a 3D array, where the first dimension corresponds to the lap number, the second dimension corresponds to the different types of information collected at each lap, and the third dimension corresponds to the different features within each type of information, we can feed this data into our neural network model. The use of a 3D dataset is necessary for the neural network to be able to learn the temporal relationships between the different input features and how they relate to the tyre changes over time. This approach allows the model to capture the patterns and trends in the data, which in turn enables accurate predictions of tyre strategies during a race.

The first step is to create a 2D dataset and then reshape it into a three-dimension. To create a dataset for all the races and drivers throughout the year, we need to generalize the method we used to create the dataset for a single race and driver. Three main steps are important in this situation:

1. *Encode strings*: encode strings such as driver names, race names, and compound names into integer values. This is important because neural networks are designed to work with numerical inputs, so we need to convert these string values to numerical values for the model to be able to process them.

2. *Replace NaN values*: the dataset may contain missing values, which are represented as NaN (Not a Number). Missing data can be problematic for some machine

learning algorithms, as they may not be able to handle NaN values. Therefore, we need to replace NaN values with some other value to ensure that the dataset is complete and usable. In this project, I replaced NaN values with -1.

3. *Same shape*: to ensure that each numpy array in the dataset has the same length. This is important because neural networks require inputs to have a fixed shape. To achieve this, we can set a fixed length for each numpy array and pad the arrays with a default value, such as -1, to fill in the empty spaces. The fixed length in this case is 78, Because the Monaco Grand Prix has the most laps of all Grand Prix.

The following is the code to get what I just described:

```
1  def load_dataset(year_list):
2  driver_race_data_list = []
3  driver_encoding = {}
4  race_encoding = {}
5  compound_encoding = {}
6
7  for year in year_list:
8      # Get the race list for the input year
9      race_list = get_race_list(year)
10
11     driver_race_data = {}
12
13     for race in race_list:
14         session = ff1.get_session(year, race, 'R')
15         session.load()
16         driver_list = pd.unique(session.laps['Driver'])
17
18         for driver in driver_list:
19             session_driver = session.laps.pick_driver(driver)
20
21             # Load all the driver's information for the current
                   session
22             data = get_data(driver, session)
23
24             # Encode and replace driver data.
25             driver_encoding[driver] = dict_data.drivers[driver]
26             driver_encoded = driver_encoding[driver]
27             data['Driver'] = data['Driver'].replace(driver,
                   driver_encoded)
28
```

```
29                  # Encode and replace race data.
30                  race_encoding[race] = dict_data.races[race]
31                  race_encoded = race_encoding[race]
32                  data['Race'] = data['Race'].replace(race, race_encoded)
33
34                  # Compound's driver data from fastf1 library.
35                  compound_list = session_driver['Compound']
36
37                  for compound in compound_list:
38
39                      # Encode and replace compound data.
40                      compound_encoding[compound] = dict_data.compound.get(
                            compound, -1)
41                      compound_encoded = compound_encoding[compound]
42                      data['Compound'] = data['Compound'].replace(compound,
                            compound_encoded)
43
44                      driver_race_data[(driver_encoded, race_encoded)] =
                            data.values
45
46                      # Add rows until lap is equal to 78 (Monaco's grand
                            prix lap).
47                      while(driver_race_data[(driver_encoded, race_encoded)
                            ].shape[0] < 78):
48                          lap = driver_race_data[(driver_encoded,
                                race_encoded)].shape[0] + 1
49                          new_row = np.array([[driver_encoded, race_encoded,
                                 lap, -1, -1, -1, -1, -1, -1, -1]])
50                          driver_race_data[(driver_encoded, race_encoded)] =
                                 np.vstack((driver_race_data[(driver_encoded,
                                race_encoded)], new_row))
51
52          # Replace NaN values with -1
53          for key, value in driver_race_data.items():
54              driver_race_data[key] = np.nan_to_num(value, nan=-1)
55
56          driver_race_data_list.append(driver_race_data)
57
58      return driver_race_data_list
```

This method returns a list of dictionaries. Each dictionary corresponds to each year. As you can see in line 9, the race list is created from a method `get_race_list(year)` which takes a year in input. This operation is necessary since the Grand Prix list changes

throughout the year.

```
def get_race_list(year):
    grand_prix_list = ff1.get_event_schedule(year)
    race_list = []

    for race in grand_prix_list['Location']:
        race_list.append(race)

    # Removing Pre-season test sessions.
    if year == 2022:
        race_list.remove('Spain')
        race_list.remove('Bahrain')

    elif year == 2021:
        race_list.remove('Sakhir')

    elif year == 2020:
        race_list.remove('Montmelo')
        race_list.remove('Montmelo')

return race_list
```

The final step in creating the dataset for predicting the optimal tyre strategy in Formula 1 races involves generating a 3D matrix that will serve as the input data for the neural network model. This matrix has dimensions $N \times T \times n$, where N represents the total number of drivers times the number of races, T represents the maximum number of laps during the race, which is 78, and n represents the number of features (such as lap time, tyre compound, and driver position) at each lap. A 3D matrix is important to train a neural network because it allows the network to learn the temporal dependencies and patterns in the data. In many real-world problems, including the case of Formula 1 tyre prediction, the input data is not just a simple vector or matrix, but rather a sequence of observations over time. By representing the data as a 3D matrix, where the first dimension is the number of samples, the second dimension is the temporal dimension, and the third dimension is the number of features, we can effectively capture the temporal dependencies in the data and enable the network to learn how to make predictions based on the history of the data. This is particularly important for problems where the input data evolves over time, such as in the case of Formula 1 races, where the lap times, tyre compounds, and other variables change over the course of the race.

Let's see the code:

```python
def generate_dataset(year_list):

    driver_race_data_list = load_dataset(year_list)

    # Determine the shape of the 3D numpy array
    m, n = next(iter(driver_race_data_list[0].values())).shape
    N = sum(len(d) for d in driver_race_data_list)
    full_dataset = np.zeros((N, m, n))

    # Convert each dictionary to a 3D numpy array and stack them
    i = 0
    for dataset in driver_race_data_list:
        for key, value in dataset.items():
            full_dataset[i] = value
            i += 1

    # Save the full dataset to a file
    np.save('ex1_data.npy', full_dataset)

return full_dataset
```

The `ex1_data.npy` is now ready to be trained into a neural network. In the following sections, we will see the details.

3.2.2 Second Experiment

The second experiment is the real project of this thesis. To predict the optimal tyre strategy, we need to have a target output y for each input in the dataset. In this case, the target output is the best tyre at a given time $t \in [0, T]$. To determine the best tyre at each time t, we use lap time as metric $\mathcal{M}$. Lap time is the amount of time it takes a driver to complete one lap of the circuit, and it is a widely used metric to measure a driver's performance during a race. However, lap time can be affected by many factors such as track conditions, weather, fuel load, traffic, and driver's performance. Therefore, it is not always accurate to say that the driver with the fastest lap time has the best tyre on. Nonetheless, for the purpose of this project, we assume that the driver with the fastest lap time has the best tyre on at that time t. We use lap time as a proxy for tyre performance, as it is the most straightforward metric to use to evaluate a driver's performance during a race.

We aim to create a dataset to predict the best compound for each time t, which is crucial for creating the best tyre strategy throughout the race. Unlike the first experiment, the information taken at each lap is different and not correlated to the driver since it does not affect the target, which is the best compound. In this case, we only focus on weather conditions, such as track temperature, rainfall, humidity, wind, and air temperature. This is because weather conditions have a significant impact on tyre performance and degradation, which can affect the choice of the next compound to pit on the car. The code and implementation for this experiment are different from the previous one since we are dealing with different information. Let's see the `pandas` dataframe creation:

```
1      def get_information(session, race, year):
2          # Get lap number for the race
3          lap = dict_data.laps[race]
4
5          # Weather conditions data
6          air_temperature = session.laps.get_weather_data()['AirTemp']
7          humidity = session.laps.get_weather_data()['Humidity']
8          pressure = session.laps.get_weather_data()['Pressure']
9          rainfall = session.laps.get_weather_data()['Rainfall']
10         rainfall = np.where(rainfall == True, 1, 0)
11
12         track_temperature = session.laps.get_weather_data()['TrackTemp']
13         wind_direction = session.laps.get_weather_data()['WindDirection']
14         wind_speed = session.laps.get_weather_data()['WindSpeed']
15
16         year_list = [year] * lap
17         race = [session.event['Location']] * lap
18
19         lap_list = []
20         for i in range(lap):
21             lap_list.append(i)
22
23         list_of_tuples = list(zip(race, year_list, lap_list,
               air_temperature, humidity, pressure, rainfall,
               track_temperature, wind_direction, wind_speed))
24
25         df = pd.DataFrame(list_of_tuples, columns = ['Race', 'Year', 'Lap'
               , 'Air Temperature', 'Humidity', 'Pressure', 'Rainfall', 'Track
                Temperature', 'Wind Direction', 'Wind Speed'])
26
27     return df
```

As you can see, the code structure is similar to the first experiment. We will still be using the same approach of creating pandas dataframe X, but the information contained within that dataframe will be different. Instead of driver information, the dataframe will contain weather data at each lap. Similarly, the target variable y will be the best tyre compound at each lap.

As we said before, the best tyre at each time t is determined based on the metric $\mathcal{M}$, which is the driver's fastest lap time per lap. To be specific, the lap times of each driver on each lap are recorded and then the tyre with the fastest lap time on that lap is identified. This process is repeated for each lap and for each driver in the race. Let's see how 2D dataset is created:

```
1  def populate_dataset(year_list):
2      dataset = []
3      race_encoding = {}
4      compound_encoding = dict_data.compound
5
6      for year in year_list:
7          # Get the race list for the current year
8          race_list = get_race_list(year)
9
10         dataset_data = {}
11
12         for race in race_list:
13             session = ff1.get_session(year, race, 'R')
14             session.load()
15
16             # Get driver's information for the current session
17             driver_information = get_information(session, race, year)
18
19             # Encode and replace race data
20             race_encoding[race] = dict_data.races[race]
21             race_encoded = race_encoding[race]
22             driver_information['Race'] = driver_information['Race'].
                   replace(race, race_encoded)
23
24             # Initialize lap data array for current race and year
25             lap_data_array = np.full((len(driver_information['Lap']), 11),
                    -1, dtype=np.float32)
26
27             for i, lap in enumerate(driver_information['Lap']):
28                 lap_data = list(driver_information.loc[driver_information[
```

```
                    'Lap'] == lap].values[0])
29                  target = get_compound_for_time(session, lap) # Best
                        compound at each lap
30
31                  if target is not None:
32                      compound_encoded = compound_encoding[target] #
                            Encoding the compound from string to integer
33                  elif target is np.NaN:
34                      compound_encoded = -1
35                  else:
36                      compound_encoded = -1
37
38                  lap_data.append(compound_encoded)
39                  lap_data_array[i,:] = lap_data
40
41              dataset_data[(race_encoded, year)] = lap_data_array
42
43              while dataset_data[(race_encoded, year)].shape[0] < 78:
44                  lap = dataset_data[(race_encoded, year)].shape[0] + 1
45                  new_row = np.array([[race_encoded, year, lap, -1, -1, -1,
                        -1, -1, -1, -1, -1]])
46                  dataset_data[(race_encoded, year)] = np.vstack((
                        dataset_data[(race_encoded, year)], new_row))
47
48          dataset.append(dataset_data)
49
50      return dataset
```

The best compound is calculated at line 29 and, after that, added in the dataset's last position.

At this point, the final step is to reshape the dataset in three-dimension.

```
1   def get_dataset(year_list):
2
3       dataset_dict = populate_dataset(year_list)
4
5       # Determine the shape of the 3D numpy array
6       m, n = next(iter(dataset_dict[0].values())).shape
7       N = sum(len(d) for d in dataset_dict)
8       full_dataset = np.zeros((N, m, n))
9
10      # Convert each dictionary to a 3D numpy array and stack them
```

```
    i = 0
    for dataset in dataset_dict:
        for key, value in dataset.items():
            full_dataset[i] = value
            i += 1


    # Save the full dataset to a file
    np.save('exp2_final_data.npy', full_dataset)

    return full_dataset
```

The final dataset `exp2_final_data.npy` is ready to be trained.

3.2.3 Neural network models

The final step of implementation is the creation and training of a neural network model. In this section, we are going to see the three neural network architectures used: the *Long Short-Term Memory* (LSTM), the *Gated Recurrent Unit* (GRU) and a *Multilayer Perceptron* (MLP). As we have seen in Chapter 2, LSTM and GRU are a type of recurrent neural network that is well-suited to processing and predicting time series data.

LSTM and GRU Model

In this code, LSTM and GRU model is implemented using the Keras library, which is a high-level neural network API written in Python and capable of running on top of various backends such as TensorFlow. The dataset used in this code is loaded from a NumPy file, from the `get_dataset()` method above, using `np.load()`. The `train_split()` function is defined to split the dataset into training and test sets. This function takes in the input data `X` and output data `y` along with a percentage `p` that determines the split ratio. The first `p` percentage of the data is used for training, and the remaining `1 - p` percentage is used for testing. The function returns two tuples, one for the training data and one for the testing data, each containing the input data and output data. It is necessary to split the dataset into training and test sets in order to evaluate the performance of the neural network model and to avoid overfitting. During the training process, the neural network learns from the training data and tries to fit the training data as closely as possible. However, if the model is overfitting the training data, it will perform poorly on new, unseen data. To avoid overfitting and to evaluate the performance of the model on new data, we use a test set. The test set is a subset of the

original dataset that is not used during training. The trained model is then evaluated on the test set to determine its accuracy and generalization performance. By evaluating the model on the test set, we can get an estimate of the model's performance on new, unseen data. This helps us to determine if the model is overfitting or underfitting and to make any necessary adjustments to the model architecture or training process.

The model that I used consists of an LSTM neural network architecture with four LSTM layers, one input layer, one output layer, and some dropout layers to reduce overfitting. Let's see now it in more details:

1. *Input layer*: it is the first layer of the neural network that receives the input data. In this case, the input layer is an LSTM layer with 128 neurons, which takes in the input data with the shape of `(x_train.shape[1:])` and applies the rectified linear unit (ReLU) activation function. The return_sequences parameter is set to True to return the entire sequence of output values.

2. *Dropout layer*: it is used to prevent overfitting. In this model, a dropout layer is added after the input layer with a dropout rate of 0.2, meaning that 20% of the input units are randomly set to 0 during each training iteration.

3. *Hidden layers:* they are responsible for learning the relevant patterns in the input data. In this model, there are three LSTM layers, each with 128 neurons and using the ReLU activation function. The return_sequences parameter is set to True for each of these layers to return the entire sequence of output values.

4. *Output layer:* it is another LSTM layer with 5 neurons, representing the 5 different categories of tyre available. The activation function used in this layer is softmax, which converts the output values to a probability distribution over the 5 categories.

After the model architecture is created, the model needs to be trained on the training dataset to learn the underlying patterns and relationships between the input data and the output data. During the training process, the model adjusts its internal parameters (weights) based on the difference between the predicted output and the true output. The process of adjusting the weights is done iteratively, with the goal of minimizing the difference between the predicted output and the true output. To train the model, we need to specify a few parameters. Let's see them:

1. *Loss function:* this function calculates the difference between the predicted output and the true output. The goal of the training process is to minimize the value of the loss function. In this code, the mean squared error (MSE) loss function is used.

2. *Optimizer:* this algorithm is responsible for updating the weights in the neural network based on the gradients of the loss function with respect to the weights. In this code, the Adam optimizer is used. The optimizer takes the learning rate as an input parameter. As we have seen in Chapter 2, the learning rate is a hyperparameter that controls the step size at which the neural network optimizer adjusts the weights during training. The optimizer tries to minimize the loss function by iteratively adjusting the weights based on the gradients of the loss function with respect to the weights.

   If the learning rate is too high, the optimizer may take large steps and overshoot the optimal weights. This can cause the model to converge slowly or not at all. On the other hand, if the learning rate is too low, the optimizer takes small steps and may converge slowly, especially if the loss function has many local minima. The learning rate schedule is often used to adjust the learning rate during training. For example, the learning rate can be reduced over time to improve convergence or increased if the training is not progressing fast enough.

   In terms of overfitting, a high learning rate can cause the model to overfit the training data. This is because a high learning rate can cause the optimizer to overshoot the optimal weights and converge to a solution that fits the training data well but does not generalize well to new data. On the other hand, a low learning rate can cause the model to underfit the training data and generalize poorly to new data. Therefore, choosing an appropriate learning rate is important to avoid overfitting and underfitting.

3. *Batch size:* during training, the model processes the data in batches. The batch size is the number of samples that are processed at once before updating the weights. In this code, the default batch size is used.

4. *Number of epochs:* An epoch is one complete pass through the entire training dataset. The number of epochs specifies how many times the training process should iterate over the entire dataset. In this code, I used a different number of epochs to see the network's behavior.

During training, the model updates its internal weights using backpropagation. The training process continues for the specified number of epochs, and after each epoch, the model is evaluated on the validation set. The validation set is a separate subset of the training data that is used to evaluate the performance of the model on data it has not seen during training. If the model is overfitting to the training data, the validation loss will start to increase while the training loss continues to decrease. In this code, the

training process is stopped early if the loss on the training set does not improve after a specific number of epochs. Normally I've set it as 20% of the epoch's total number.

The following code is how LSTM has been implemented. It's important to say that GRU has been implement at the same way. Only Keras object is different.

```
dataset = np.load('exp2_final_data.npy')

X = dataset[:, :, :-1]
y = dataset[:, :, -1]

def train_split(X, y, p):
    N = len(X)
    n_train = int(N*p)

    x_train = X[:n_train]
    y_train = y[:n_train]
    x_test = X[n_train:]
    y_test = y[n_train:]

    return (x_train, y_train), (x_test, y_test)

(x_train, y_train), (x_test, y_test) = train_split(X, y, 0.8)
y_train = tf.keras.utils.to_categorical(y_train, 5)
y_test = tf.keras.utils.to_categorical(y_test, 5)

model = Sequential()

#------------- Input layer -------------
model.add(LSTM(128, input_shape = (x_train.shape[1:]), activation = '
    relu', return_sequences = True))
model.add(Dropout(0.2))

#------------- Hidden layers -------------
model.add(LSTM(128, activation = 'relu', return_sequences = True))
model.add(LSTM(128, activation = 'relu', return_sequences = True))
model.add(LSTM(128, activation = 'relu', return_sequences = True))

# ------------- Output layer -------------
model.add(LSTM(5, input_shape = (x_train.shape[1:]), activation = '
    softmax', return_sequences = True))

opt = tf.keras.optimizers.Adam(learning_rate = 1e-4)
model.compile(loss = 'mse',
```

```
37                      optimizer = opt,
38                      metrics = ['accuracy'])
39
40      # Checkpoint and early stop
41      checkpoint = ModelCheckpoint("model_weight_lstm.h5", save_best_only=
           True, save_weights_only=True, monitor='loss', mode='min', verbose
           =1)
42      early_stop = EarlyStopping(monitor='loss', patience=400, mode='min',
           verbose=1)
43
44      # Train the model
45      time_callback = TimeHistory()
46      hist = model.fit(x_train, y_train, validation_data = (x_test, y_test),
            epochs = 2000, shuffle = False, callbacks=[time_callback,
           checkpoint, early_stop])
47
48      score = model. Evaluate(x_test, y_test, verbose=0)
49
50      print('Test loss:', score[0])
51      print('Test accuracy:', score[1])
```

MLP Model

The last network used is the *Multi-Layer Perceptron* (MLP). We decided to use a non-recurrent neural network to see his behavior with this type of project. Theoretically, it isn't the best neural network to use since it doesn't consider the temporal aspect of the data. The results will be shown in the next chapter. The main difference with the GRU model is the input data. It's flattened into a 2D array and each time step is treated as an independent input. `Flatten` and `reshape` are methods used to manipulate the shape of an array. In the given code, `X` and `y` are reshaped to be fed into the MLP. Specifically, `X` is a 3D array with shape `(78, 78, 5)`, where 5 is the number of features. However, the MLP expects 2D input, hence `X` is reshaped into a 2D array with shape `(6084, 5)` using the reshape method. Similarly, `y` is flattened from a 2D array with shape `(78, 78)` to a 1D array with shape `(6084,)`. This is because the MLP expects a 1D array of labels.

In an MLP, each sample (where sample means each row of the dataset with all the information, i.e. a particular combination of race, year, lap, and weather conditions) is represented as a one-dimensional feature vector. Therefore, the input data for an MLP is usually represented as a two-dimensional array, where the first dimension corresponds to

the number of samples, and the second dimension corresponds to the number of features in each sample. In contrast, GRU is a type of RNN that is designed to process sequences of data, where each data point can be a vector or a sequence itself. Therefore, the input data for a GRU is usually represented as a three-dimensional array, where the first dimension corresponds to the number of sequences, the second dimension corresponds to the length of each sequence, and the third dimension corresponds to the number of features in each data point. Regarding the output, in an MLP, the model predicts a single output value for each sample, so the target variable (y) is usually represented as a one-dimensional array. On the other hand, in GRU, the model can predict an output sequence for each input sequence, so the target variable (y) can also be represented as a two-dimensional array, where the first dimension corresponds to the number of sequences, and the second dimension corresponds to the length of each output sequence.

The MLP also has a simpler architecture with only fully connected layers, while the GRU had recurrent layers that allowed it to capture temporal dependencies in the data.

```
1  dataset = np.load('exp2_final_data.npy')
2
3  X = dataset[:, :, :-1]
4  y = dataset[:, :, -1]
5
6  X = X.reshape((78*78, -1))
7  y = y.flatten()
8
9  def train_split(X, y, p):
10     N = len(X)
11     n_train = int(N*p)
12
13     x_train = X[:n_train]
14     y_train = y[:n_train]
15     x_test = X[n_train:]
16     y_test = y[n_train:]
17
18     return (x_train, y_train), (x_test, y_test)
19
20 (x_train, y_train), (x_test, y_test) = train_split(X, y, 0.8)
21 y_train = tf.keras.utils.to_categorical(y_train, 5)
22 y_test = tf.keras.utils.to_categorical(y_test, 5)
23
24 # --------------------- Model ---------------------
25 model = Sequential()
26
```

```
27 # --------------------- Fully Connected Layers ---------------------
28 model.add(Dense(128, activation = 'relu', input_dim = X.shape[-1]))
29 model.add(Dense(64, activation='relu'))
30 model.add(Dense(64, activation = 'relu', input_dim = X.shape[-1]))
31
32 # --------------------- Output layer ---------------------
33 model.add(Dense(5, activation='softmax'))
34
35 model.compile(loss = 'mse', optimizer = 'adam', metrics = ['accuracy'])
36
37 # Checkpoint and early stop
38 checkpoint = ModelCheckpoint("model_weight_lstm.h5", save_best_only=True,
       save_weights_only=True, monitor='loss', mode='min', verbose=1)
39 early_stop = EarlyStopping(monitor='loss', patience=400, mode='min',
       verbose=1)
40
41 # Train the model
42 time_callback = TimeHistory()
43 hist = model.fit(x_train, y_train, validation_data = (x_test, y_test),
       epochs = 2000, shuffle = False, callbacks=[time_callback, checkpoint,
       early_stop])
44
45 score = model.evaluate(x_test, y_test, verbose=0)
46 print('Test loss:', score[0])
47 print('Test accuracy:', score[1])
```

In this code, the MLP model consists of three fully connected layers with ReLU activation functions and a softmax output layer. The input layer has 128 neurons, and the two hidden layers have 64 neurons each. The output layer has five neurons, which represent the five different classes. The model is compiled using mean squared error as the loss function, the Adam optimizer, and accuracy as the evaluation metric. The code, as you can see, it's quite similar to LSTM and GRU, even if the differences are a lot.

This is the code implementation. In the next chapter we are going to see the results obtained and which is the best neural network for this project.

Chapter 4

Numerical Results

In this chapter, we are going to see the model's results and we will compare them with each other. We will focus only on the second experiment since it's the aim of the project itself. We will discuss two points:

1. Compare LSTM with GRU
2. Compare GRU with MLP

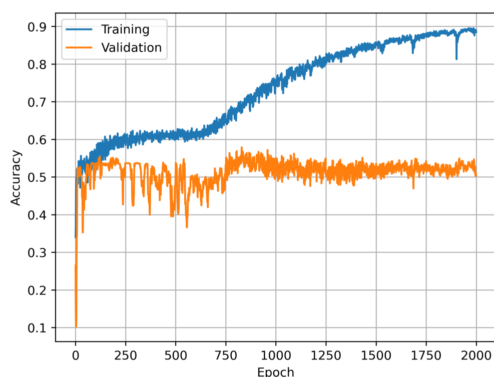
4.1 LSTM vs GRU

As we have seen in the previous chapter, the choice of hyperparameters is critical in determining the performance of the model. One of the most important hyperparameters is the learning rate, which controls the step size at which the model parameters are updated during training. To investigate the effect of different learning rates on the performance of the LSTM and GRU models, we trained each model with two different learning rates, namely $1e-4$ and $5e-4$. For each learning rate, we trained the models for a fixed number of epochs and recorded their accuracy and loss values on both the training and validation sets.

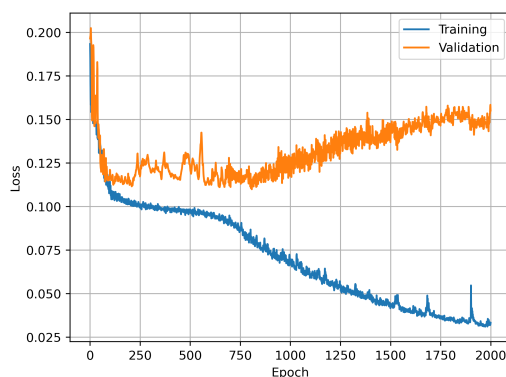
From our experiments, we observed that the GRU consistently outperformed the LSTM across all learning rates. Specifically, with a learning rate of $1e-4$, the GRU achieved an accuracy of 0.514 and a loss of 0.118, while the LSTM only achieved an accuracy of 0.238 and a loss of 0.160. Similarly, with a learning rate of $5e-4$, the GRU achieved an accuracy of 0.504 and a loss of 0.155, while the LSTM achieved an accuracy of 0.281 and a loss of 0.156. These results suggest that the GRU is better suited for this particular task than the LSTM, as it consistently achieved higher accuracy and lower

loss values across all learning rates. Moreover, we observed that increasing the learning rate did not necessarily lead to better performance, as the LSTM model achieved its best accuracy with a lower learning rate of $1e-4$.

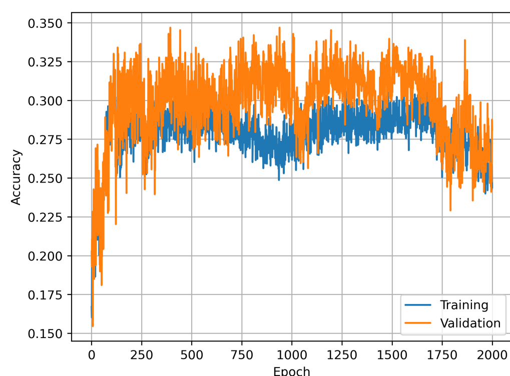
In conclusion, our experiments demonstrate the importance of choosing appropriate hyperparameters in determining the performance of the model and provide evidence that the GRU is a more suitable architecture for this particular task than the LSTM.



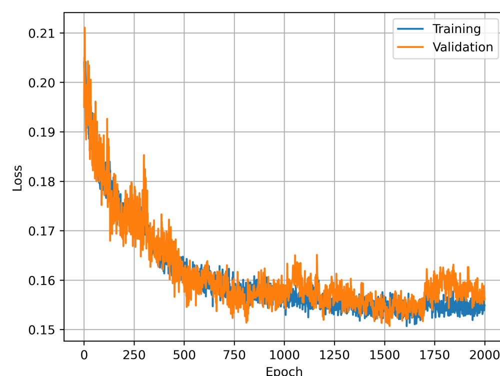
(a) GRU Accuracy



(b) GRU Loss

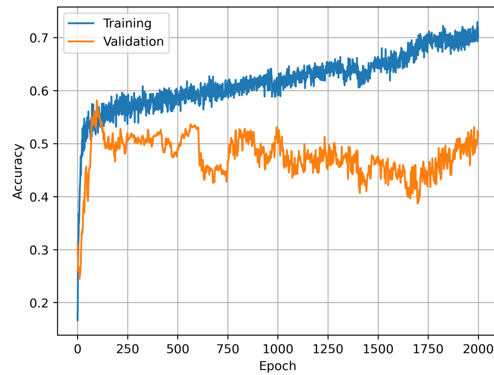


(c) LSTM Accuracy

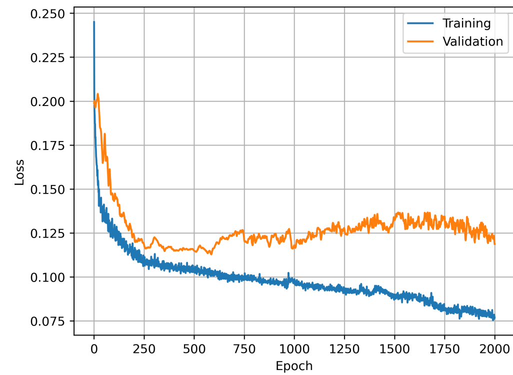


(d) LSTM Loss

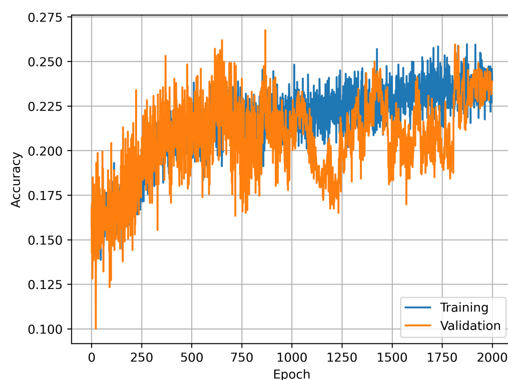
Figure 4.1: The figure shows the training and validation accuracy of two neural network models, LSTM and GRU, with a learning rate of 5e-4. The graph displays a comparison of the performance of the two models over time. The x-axis shows the number of epochs, while the y-axis shows the accuracy. The blue line represents the training accuracy, while the orange line represents the validation accuracy.



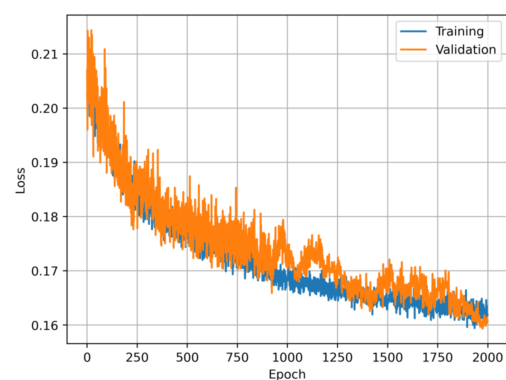
(a) GRU Accuracy



(b) GRU Loss



(c) LSTM Accuracy



(d) LSTM Loss

Figure 4.2: The figure shows the training and validation accuracy of two neural network models, LSTM and GRU, with a learning rate of 1e-4. The graph displays a comparison of the performance of the two models over time. The x-axis shows the number of epochs, while the y-axis shows the accuracy. The blue line represents the training accuracy, while the orange line represents the validation accuracy.

As you can see in Figure 4.1 and Figure 4.2, GRU outperformed LSTM in terms of accuracy and loss. GRU networks are particularly useful for processing sequential data due to their ability to handle longer-term dependencies. In this particular project, the weather conditions information for every single lap in the race for each year is a sequence of data that needs to be analyzed in order to make predictions. GRU is able to handle this type of sequence data well because of the way it is designed. One of the reasons

why GRU works better in this project compared to LSTM is because it has a more efficient gating mechanism. The GRU has two gates, the update gate and the reset gate, which allow it to selectively discard or pass on information to the next time step. This mechanism enables GRU to capture longer-term dependencies without suffering from the vanishing gradient problem that can occur in LSTM networks.

Another reason why GRU is performing better could be due to the size of the dataset. The GRU architecture may be more efficient at capturing the relevant patterns in the data with fewer parameters compared to LSTM, which has more complex and larger architectures. Additionally, the GRU architecture can be trained faster and requires less memory compared to LSTM, which may be particularly important when dealing with large datasets. Overall, the GRU network seems to be a better choice for this project due to its ability to handle sequential data with longer-term dependencies and its more efficient gating mechanism. These factors make it a suitable architecture for capturing patterns in the dataset related to weather conditions and making accurate predictions.

4.2 GRU vs MLP

In the previous experiment, we compared the performance of two types of recurrent neural networks, LSTM and GRU. In this section, we will compare the performance of GRU with that of a different type of neural network, MLP.

The MLP model consists of three dense layers with 128, 64, and 64 neurons respectively, followed by a final output layer with 5 neurons, representing the different tyre options available. This model was trained for 2000 epochs, resulting in an accuracy of 0.271 and a loss of 0.291. On the other hand, the GRU model consists of three GRU layers with 128 units each, followed by a dense layer with 5 neurons for the output. The GRU model was also trained with the same data and for the same number of epochs as the MLP model. The GRU model outperformed the MLP model with an accuracy of 0.514 and a loss of 0.118 with a learning rate of $1e-4$. When the learning rate was increased to $5e-4$, the accuracy dropped slightly to 0.504, but still outperformed the MLP model with a loss of 0.155.

The superior performance of the GRU model can be attributed to its recurrent nature, which allows it to better capture the sequential nature of the data, considering the previous tyre choices and lap times to make a prediction for the current lap. This is particularly important in the case of predicting tyre strategy, as it requires taking into account not only the current lap conditions but also the previous choices made by the driver and the team.

In summary, the GRU model proved to be a more effective approach for predicting

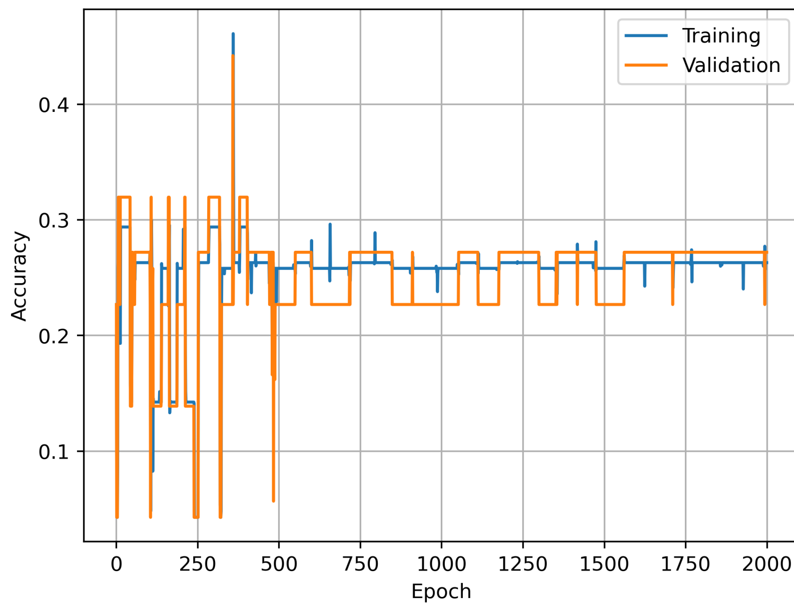


Figure 4.3: The figure depicts the training and test accuracy of an MLP (Multi-Layer Perceptron) model. The graph displays the performance of the model over time, with the x-axis showing the number of epochs and the y-axis showing the accuracy. The blue line represents the training accuracy, while the orange line represents the test accuracy.

the best tyre strategy for Formula 1 drivers. The superior performance of the GRU model can be attributed to its ability to capture the sequential nature of the data, which is particularly important in this case due to the strategic nature of the tyre choices during a race. In Figure 4.3 is possible to see the MLP's accuracy during training and testing. It's clear that MLP doesn't learn at all. That's probably because its nature isn't recurrent.

4.3 Blind Classifier

In machine learning, a blind classifier is a model that predicts the class label of the test data based only on the prior probabilities of the classes in the training data. It is called "blind" because it does not use any information about the input features, and simply relies on the frequency of the classes in the training data.

```
1 dataset = np.load('exp2_final_data.npy')
```

```
2
3 X = dataset[:, :, :-1]
4 y = dataset[:, :, -1]
5
6 X = X.reshape((78*78, -1))
7 y = y.flatten()
8
9 def train_split(X, y, p):
10     N = len(X)
11     n_train = int(N*p)
12
13     x_train = X[:n_train]
14     y_train = y[:n_train]
15     x_test = X[n_train:]
16     y_test = y[n_train:]
17
18     return (x_train, y_train), (x_test, y_test)
19
20 (x_train, y_train), (x_test, y_test) = train_split(X, y, 0.8)
21 y_train = tf.keras.utils.to_categorical(y_train, 5)
22 y_test = tf.keras.utils.to_categorical(y_test, 5)
23
24 p_i = np.mean(y_train, axis = 0)
25 n_i = np.sum(y_test, axis = 0)
26 blind_classifier = np.sum(p_i * n_i) / np.sum(n_i)
```

In the code above, we first split the data into training and test sets, and then convert the class labels to one-hot encoded format using the `to_categorical()` function from Keras. Next, we calculate the prior probabilities of the classes in the training set using the `mean()` function along the first axis, which corresponds to the samples. Similarly, you calculate the number of samples for each class in the test set using the `sum()` function along the first axis. I followed this procedure since the formula to calculate the blind classifier is the following:

$$\text{acc} = \frac{\sum_{i=1}^{c} p_i n_i}{\sum n_i} \tag{4.1}$$

This formula gives us the expected accuracy of a blind classifier that simply predicts the most frequent class in the training set. In our case, the blind classifier accuracy is 24.5%, which means that if you were to randomly guess the class label for each test sample, you would expect to get an accuracy of 24.5%. Therefore, any machine learning model that you train should aim to perform better than this baseline accuracy, in order

to be considered useful.

The performance of the used models can be compared with the blind classifier to evaluate their effectiveness in predicting the tyre strategy. Comparing the accuracy of the three models with the blind classifier, we can see that the GRU model performed the best, achieving an accuracy of 51.4% with a learning rate of $1e-4$ and 50.4% with a learning rate of $5e-4$. The LSTM model performed worse than the GRU model, achieving an accuracy of only 23.8% with a learning rate of $1e-4$ and 28.1% with a learning rate of $5e-4$. The MLP model, on the other hand, performed the worst among the three models, achieving an accuracy of 27.1% after training for 2000 epochs. This suggests that the MLP model may not be suitable for this type of project, as its non-recurrent nature may not be powerful enough to capture the sequential patterns in the data.

Overall, the results indicate that the recurrent nature of the GRU and LSTM models makes them more effective at capturing the sequential patterns in the data and predicting the tyre strategy accurately. The superior performance of the GRU model over the LSTM model may be due to the fact that the GRU is better at handling vanishing gradients, which is a common problem in LSTM networks.

In conclusion, we have trained and compared the performance of three different models, namely GRU, LSTM, and MLP, to predict the best tyre strategy for Formula 1 races. We have observed, Figure 4.4 and Table 4.1, that the recurrent nature of GRU and LSTM makes them better suited for this type of project, compared to MLP. This is because the tyre strategy is highly dependent on the previous laps' conditions, making the use of previous information crucial. Our experiments have shown that, among the tested models, the GRU performs better.

Model	Learning Rate	Train		Test	
		Accuracy	Loss	Accuracy	Loss
MLP	-	0.287	0.304	0.271	0.291
LSTM	1e-4	0.244	0.167	0.238	0.16
	5e-4	0.278	0.159	0.282	0.156
GRU	1e-4	0.756	0.075	0.514	0.118
	5e-4	0.897	0.031	0.504	0.155

Table 4.1: Neural network models and their respective training and test accuracies and losses for different learning rates

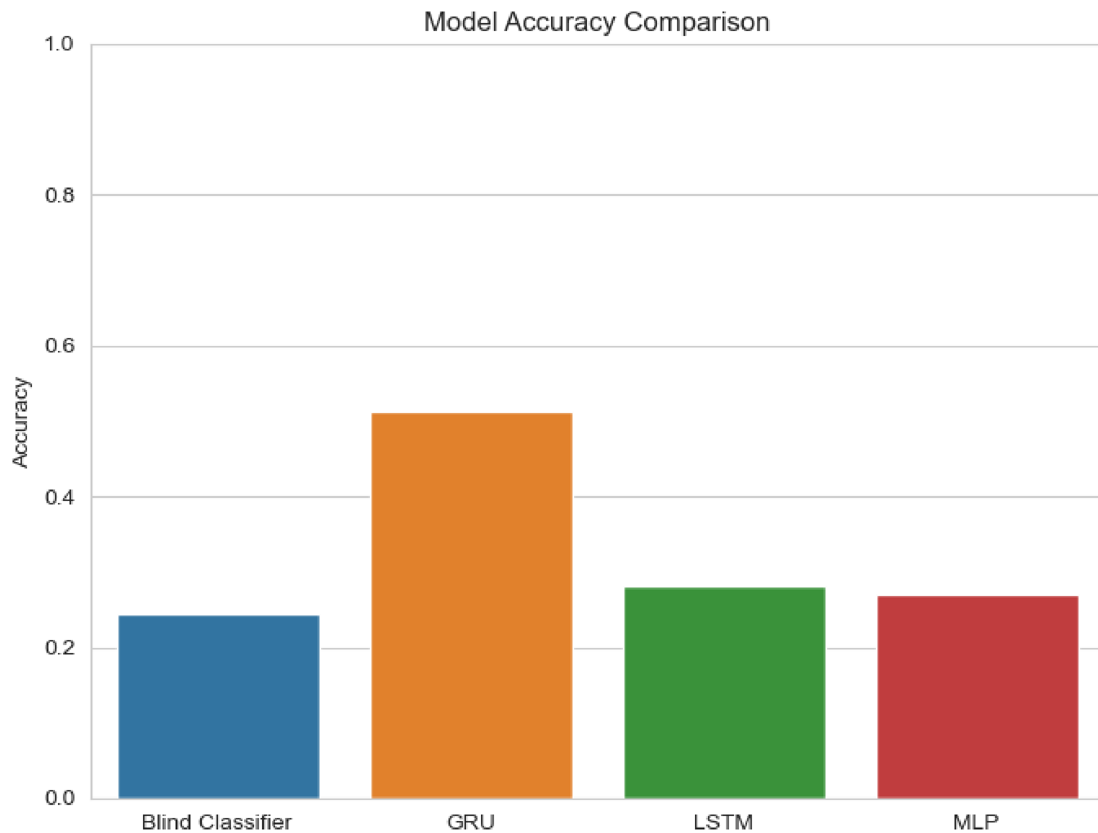


Figure 4.4: Model Accuracies Comparison between Blind Classifier, GRU, LSTM, and MLP - This figure compares the performance of four different models: Blind Classifier, GRU, LSTM, and MLP. The accuracy metric can be used to evaluate how well each model performs on the task.

Finally, we have plotted the percentage of tyre usage and tyre usage times, Figure 4.5 and Figure 4.6, to provide insight into the most common tyre strategies used in Formula 1 races. These plots could be useful for teams to determine the best tyre strategies for upcoming races.

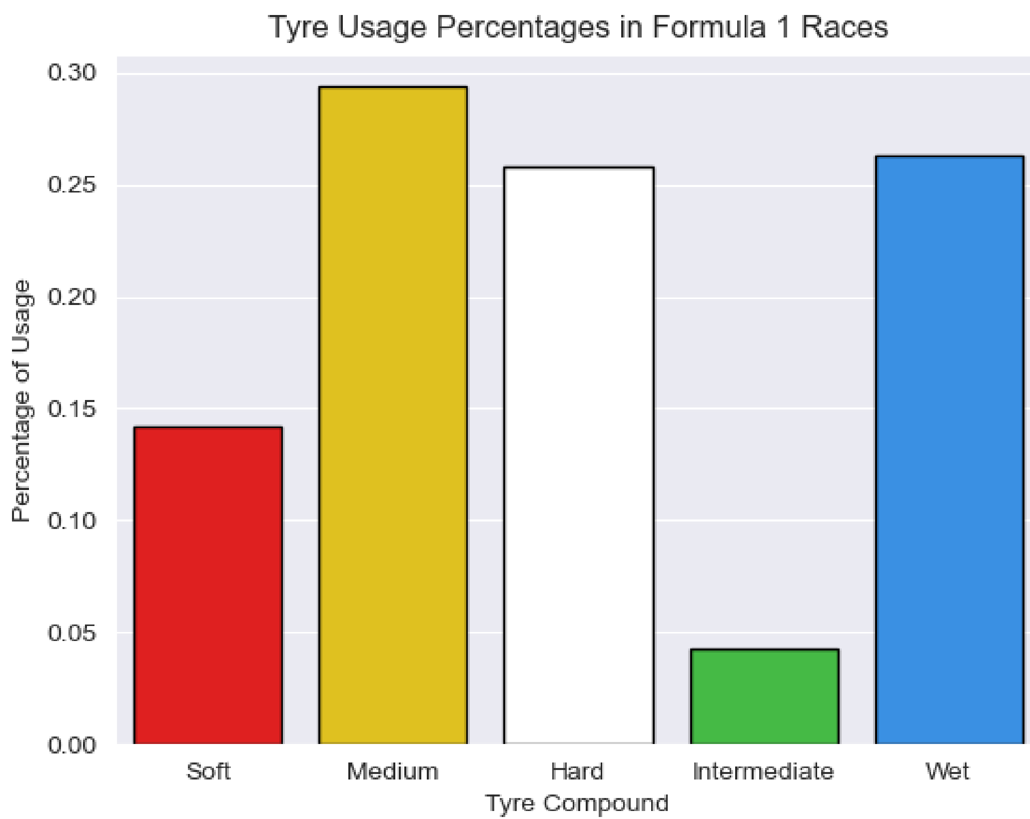


Figure 4.5: Tyre Usage Percentages - The figure displays the percentage of tyre usage during the races analyzed. The graph shows a breakdown of the percentage of usage of each tyre compound over time. The graph provides a clear visual representation of the tyre usage patterns during the races.

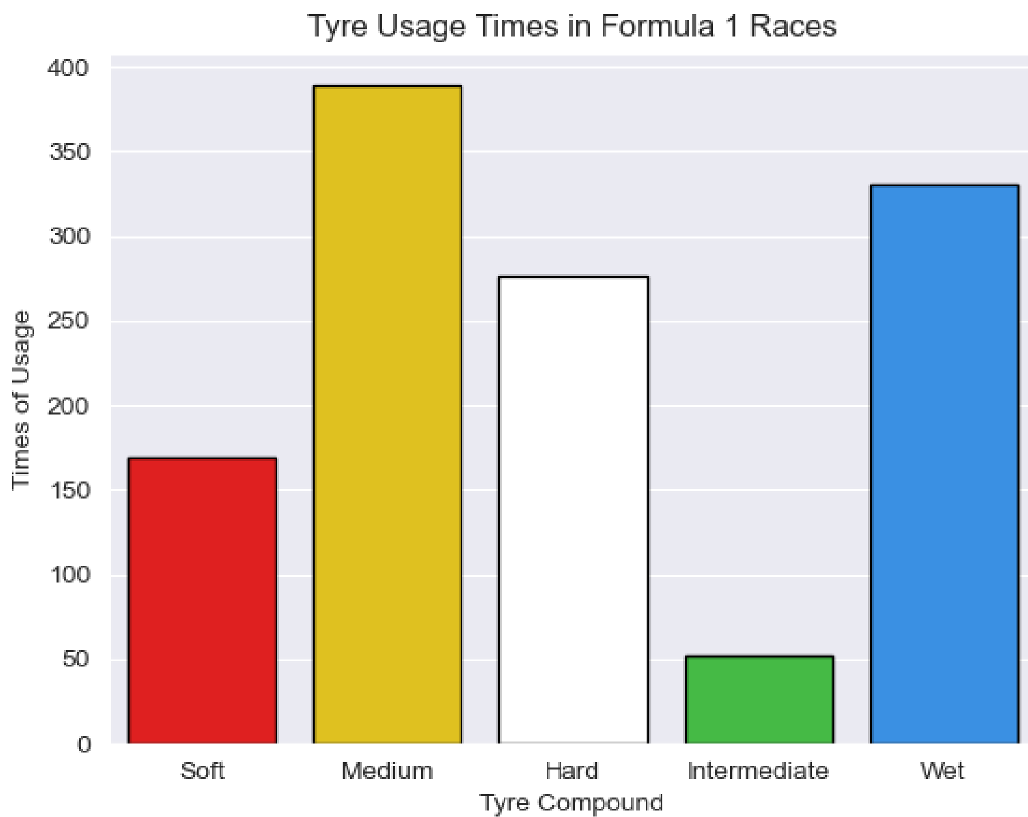


Figure 4.6: Tyre Usage Times - The figure displays the times of tyre usage during the races analyzed. The graph provides a clear visual representation of the tyre usage patterns during the races.

Chapter 5

Conclusion

In this bachelor's thesis, we have explored the potential of deep learning and neural network algorithms in predicting tyre strategy during a Formula 1 race. Our aim was to showcase the practical application of these algorithms in a high-performance environment such as Formula 1 and demonstrate their effectiveness in predicting the optimal time to change tyres and which compounds to use. Through the utilization of telemetry data sourced from the "FastF1" API and the development of LSTM, GRU, and MLP models, we were able to achieve promising results. Our models demonstrated a great accuracy in predicting the best tyre strategy, where GRU outperformed the blind classifier by 25%. These results highlight the potential of deep learning and neural network algorithms in tyre prediction and can provide valuable insights into the development of more advanced tyre prediction models in the future.

In conclusion, the use of deep learning and neural network algorithms in predicting tyre strategy during a Formula 1 race has the potential to revolutionize the sport. By analyzing large amounts of data and making informed decisions on tyre changes, teams can optimize their performance and increase their chances of success on the track. As technology continues to advance, the use of deep learning and AI will undoubtedly become more prevalent in motorsports, providing engineers with even more accurate predictions and allowing them to make the best decisions possible.

Bibliography

[1] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[2] Aurélien Géron. "Hands-on machine learning with scikit-learn and tensorflow: Concepts". In: *Tools, and Techniques to build intelligent systems* (2017).

[3] 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*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[4] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[6] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[7] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

Acknowledgements

At the end of this thesis, it seems only right to dedicate a space to thank all the people who, with their support, have helped me on this beautiful path of deepening the knowledge acquired during my university years.

First, I would like to thank my co-supervisor Davide Evangelista, for his valuable advice and for his availability. Thank you for providing me with fundamental insights in drafting this work and directing me in moments of indecision. I wanna thank my supervisor also, Elena Loli Piccolomini, to give the possibility to build the idea that I had in mind. I'm very happy about the results and all the work we've done.

I can't fail to thank the two people who have had the most influence on my educational path: my parents. Mom and Dad: thank you for helping me through the most difficult moments, without you I could never have reached this important milestone!

Thanks to my lifelong friends, Gold One, who lightened my heaviest moments and encouraged me to give more and more.

Thanks to my friends that I got to know during my university life, both in Bologna and during my experience abroad in Kaunas. We spent beautiful and very tiring moments together. Those were years I won't forget. I will miss studying on Discord with the Massoneria Siummica. It all started with the famous "Algoritmo di Gauss" in the Linear Algebra class!

Thank you.