As a start distribution π is required; setting $\pi = [0.1, 0.7, 0.2]$ for Figure 3(a) would mean a probability 0.7 of starting in cold (state 2), probability 0.1 of starting in hot (state 1), etc.

More formally, consider a sequence of state variables $q_1, q_2, \ldots, q_i$. A Markov model symbolizes the Markov assumption on the probabilities of this sequence: that when predicting the future, the past has no significance, only the present.

Markov Assumption: $$P(q_i = a|q_i, \ldots, q_{i-1}) = P(q_i = a|q_{i-1}) \quad (1)$$

Figure 3(a): Shows a Markov Chain for conveying a probability to a sequence of weather events, for which the states consists of HOT, COLD, and WARM. The states are represented as nodes in the graph, and the transitions, with their probabilities as edges. The transition is probabilities: the values of arcs leaving a given state must sum to 1. Figure 3(b) shows a Markov chain for assigning a probability to a sequence of words $w_1, \ldots, w_n$. This Markov Chain represents a bigram language model, with each edge expressing the probability $P(w_i|w_j)$. Given the two models in Figure 3, a probability to any sequence from our vocabulary can be assigned.

Formally, a Markov Chain is defined by the following components:

$Q = q_1, q_2, \ldots, q_N$ a set of N states,

$A = \begin{bmatrix} a_{11} & a_{11} & a_{1n} \\ a_{21} & a_{22} & a_{2n} \\ : & : & : \\ a_{n1} & a_{n2} & a_{nn} \end{bmatrix}$ a transition probability matrix A, each a_{ij} representing the probability of moving state i state to state j, s.t. $\sum_{j=1}^{n} (a_{ij}) = 1 \quad \forall i$

$\pi = \pi_1, \pi_2, \ldots, \pi_N$ an initial probability over states. π_i is the probability that the Markov Chain will start in state i. Some states j may have $\pi_j = 0$, referring that they cannot be initial states. Also, $\sum_{i=1}^{n}(\pi_i) = 1$

2.4 Machine Learning

The term machine learning was coined by Samuel in the 1950s. In the introduction to his work, Samuel describes how his studies were concerned with: "the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning." More than half a century later, this notion of machine learning is frequently used across multiple fields.

Machine learning (ML) is a type of algorithm, which allows software applications to become more precise in predicting outcomes without having explicitly programmed. The basic premise of ML is to build algorithms which can receive input data and use statistical analysis to predict output at the same time updating outputs as new data becomes available.

Machine learning can be divided into three subfields: supervised, unsupervised, and reinforcement learning. The techniques used in this thesis are based on supervised learning, and a rigorous explanation of the method is given in Section 2.3.1.

2.4.1 Supervised learning

In supervised learning, algorithms require labeled data to learn from. After understanding the data, the algorithm determines which label should be given to new data based on pattern and associating the patterns to the unlabeled new data.

Supervised learning mostly divided into two categories, i.e. Classification & Regression. Classification predicts a category the data belongs to, i.e., Sentiment Analysis, Spam Detection, Dog or Cat classification. Regression predicts a numerical value based on previously observed data. i.e., Stock Price Prediction, House Price Prediction. Since our thesis focuses on velocity prediction, hence, it is a Regression Analysis method is used for supervised learning.

Here's an explanation of how a regression model is used for prediction with arbitrary data (given in Table 1). Regression models have various types of model for an understanding purpose we will be using Linear Regression model.

Years of Experience	Salary per Anum (Euros)
1	51000
2	53000
3	57000
4	59000
5	61500
6	62500

Table 1: Sample values of Salary corresponding to years of experience

Linear Regression performs the task to predict a dependent variable value (y) based on a given independent variable (x). So, this regression techniques finds out a linear relationship between x (input) and y (output). Hence, the name is Linear Regression. X (input) is the work experience, and Y (output) is the salary of a person. Our task is to approximately predict salary after 8 years of work experience of a person. Hypothesis function for linear regression is $y = \theta x + \beta$ While training the model we are given:

x: input data (univariate-one input variable(parameter)), y: labels to data (supervised learning), θ: coefficient of x, β: intercept.

When training the model- it fits the optimum line to predict the value of y (salary) for a given value of x. The model gets the best regression fit line by finding the best θ and β values. Once we find the best θ and β values, we get the optimum fit line. So, when we finally use our model for prediction, it will predict the value of y (salary) for the input value of x (years of experience). In order to find the best fit line, we need to update β and θ values, to do so we will use cost function.

$$J = \frac{1}{n}\sum_{i=0}^{n}(pred_i - y_i)^2 \tag{2}$$

$$minimize\frac{1}{n}\sum_{i=0}^{n}(pred_i - y_i)^2 \tag{3}$$

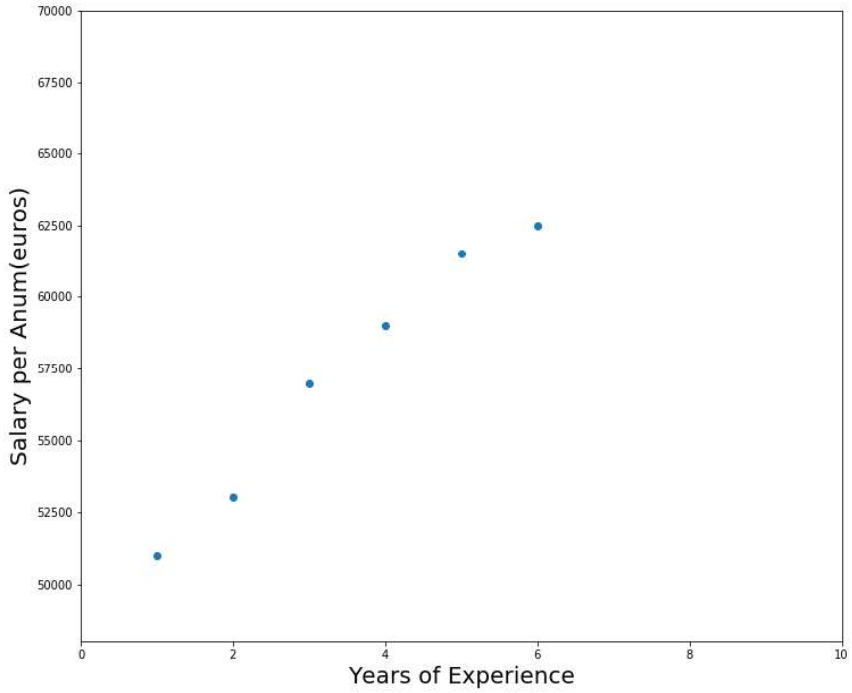

Figure 4: Shows graphical representation of Table 2.3.1.

By achieving the optimum-fit regression line, the model aims to predict y value such that the error difference between the predicted value and true value is minimum. So, it is essential to update the β and θ values, to reach the optimum value that has minimum error between true y value (y) and predicted y value (*pred*).

Cost function (J) of Linear Regression is the Root Mean Squared Error (RMSE) between the true y value (y) and predicted y value (*pred*) Equation (2)

To update β and θ values to reduce Cost function (minimizing RMSE value) and achieving the best fit line, the model uses Gradient Descent (Equation (3)). The idea is to start with random β and θ values, and then iteratively updating the values, reaching minimum cost. The best fit line is shown in Figure 5. And from Figure 5, we can obtain the expected approximately salary after having 8 years of experience would be somewhere around 68250 Euros.

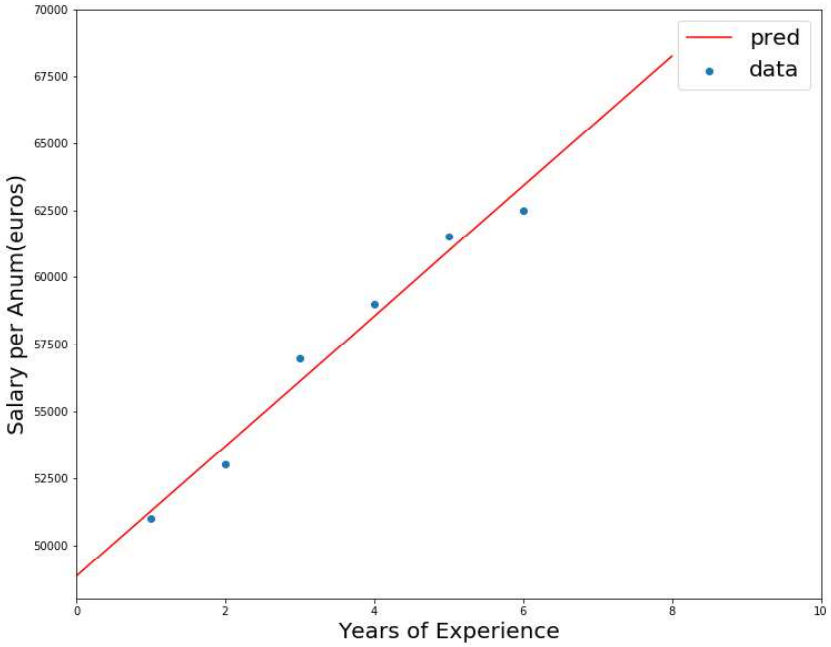

Figure 5: Shows regression line generated from Table 2.3.1 data

2.4.1.1 Time-series prediction as a supervised learning problem

Time-series do not have an obvious input/output-mapping, but it is possible to transform a time-series into a form apt for supervised learning via generating input and output sequences.

Time	Value
0	0.00
1	3.75
2	5.25
3	4.00
4	8.50

Table 2: The first five values from the speed-profile time-series shown in Figure 1

x	*y*
0.00	3.75
3.75	5.25
5.25	4.00
4.00	8.50

Table 3: The resulting table after transforming the values of the time-series into a dataset apt for supervised learning.

2.4.2 Neural Networks

Neural Networks (NN) were originally introduced for mathematically modeling of the way the biological brains process information (McCulloh and Pitts, 1943; Rosenblatt, 1962; Rumelhart et al.; 1986). Today, Neural Network can be varied in an almost infinite number of ways, but most commonly referred and basic is the Multilayer Perceptron (Vanilla Neural Network).

NN is a set of connected neurons organized in layers:

- **Input layer:** It brings the initial data into the system for further processing by subsequent layers of neurons.
- **Hidden layer:** It is a layer in between input layers and output layers, where neurons take in a set of weighted inputs and produce an output through an activation function.
- **Output layer:** It is the last layer of neurons that produces given outputs for the program.

Multilayer Perceptrons (MLP) are commonly used for classification. The MLP is a fully connected, feedforward neural network with at least three layers. Fully connected NN refers that every neuron in a layer is connected to every neuron in the next layer. In addition to this, feedforward implies that the connections in the network never cause loops nor skip any layers [15].

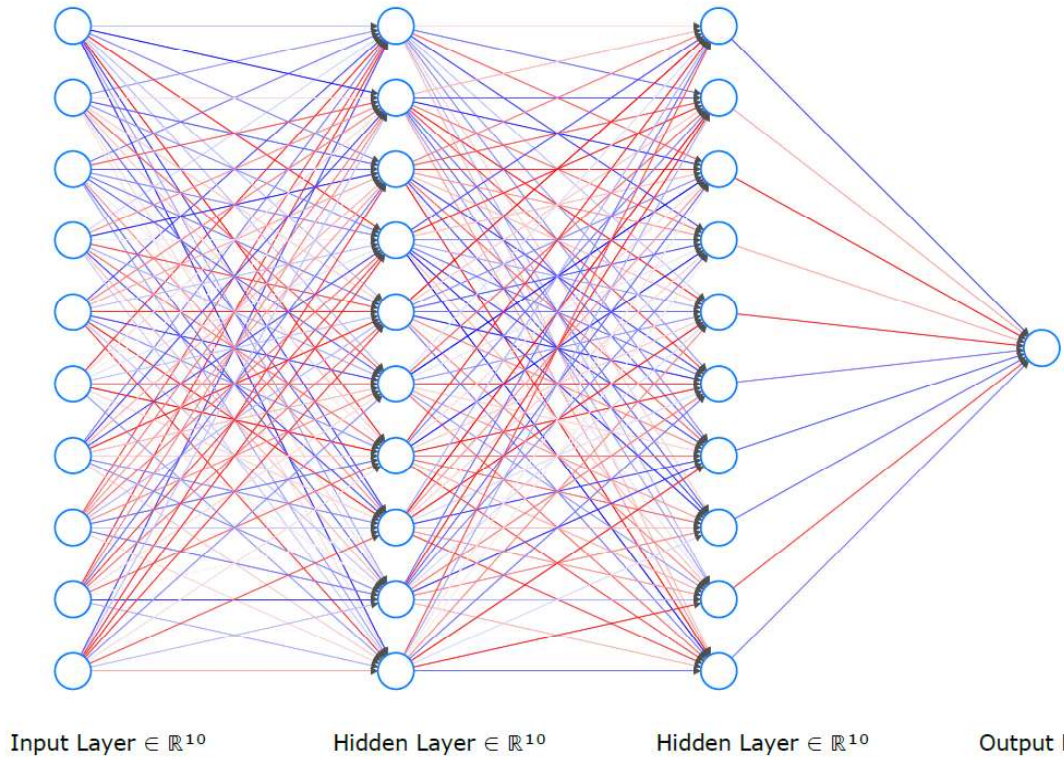

Figure 6: Shows a multilayer perceptron with ten inputs, two hidden layers (with ten neurons in each layer) and a single output neuron in its output layer.

Each line in Figure 6 represents a connection between two neurons. By assigning a weight w to each connection in the network, it is possible to control which neurons have more or less influence on the data passed forward. These connections can either inhibit or exhibit the flow of data between the neurons. In addition to the weights, it is also customary to assign a bias b to every Neuron of the MLP.

2.4.3 Training a Neural Network

All NNs require training before they are used for the application. In the beginning, the weights and biases are randomly assigned – usually in a small interval around zero; hence, the predictions of the network will be nothing but random. To achieve better accuracy or results

from NN it requires training in an iterative manner which constitutes of two steps: 1. A forward pass of data to retrieve a prediction and 2. A backward pass to update the parameters (like weights and biases) of the network – is called a "Backpropagation process."

2.4.3.1 Acquiring a prediction from a Neural Network

The forward pass of input data in MLP yields a prediction (output). Except for the neurons in the input layer, each neuron in an MLP applies a nonlinear activation function g (i.e., Sigmoid, ReLU, tanh) to a linear combination of the data given by the previous layer, as shown in Figure 7.

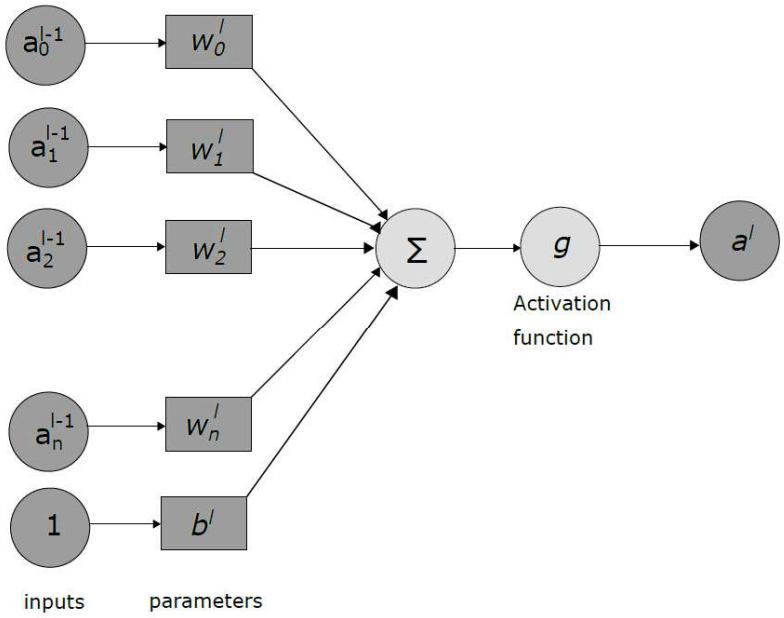

Figure 7: Shows the intervals of a Neuron in a multilayer perceptron.

The activation of the i^{th} Neuron in layer l depends on the parameters between layers l - *1* and l, as well as the activations from layer l - *1*, following Equation (4). We denote the weight for the connection between Neuron k in layer l - *1* and Neuron j in layer l as w_{jk}^l, similarly b_j^l indicates the bias of the j^{th} Neuron in the l^{th} layer. In Figure 7 For ease of understanding, we assume that layer l contains only one Neuron, why we may omit the double indices.

$$a_j^l = g\left(\sum_{k=0}^{n}\left(w_{j\ k}^l a_k^{l-1}\right) + b_j^l\right) \tag{4}$$

It is possible that the activation function g differs between layers in an MLP. The softmax function is commonly used as the activation function for the output layer, to retrieve a vector which can represent a probability distribution over some classes. However, more often the sigmoid function (Equation (5)), the hyperbolic tangent function (Equation (6)) or the rectified linear unit (Equation (7)) are used as activation functions for the neurons in the hidden layers.

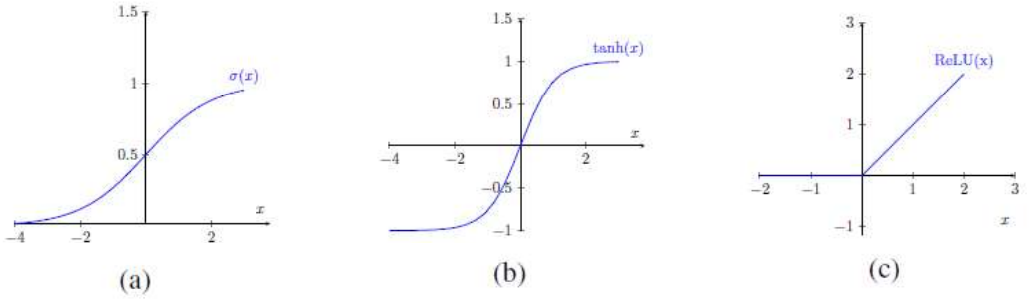

Figure 8: Shows plots of different activation functions. (a) sigmoid function, (b) hyperbolic function, (c) rectified linear unit [41].

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{5}$$

$$tanh(t) = \frac{e^{2t} - 1}{e^{2t} + 1} \tag{6}$$

$$ReLU(t) = \max(0, t) \tag{7}$$

2.4.3.2 Updating the parameters of a Neural Network

The backword pass of an MLP is performed in order to update the parameters of the NN, and in turn, lower the error L (Loss) of the network. The process executes in two steps: 1. Calculating partial derivatives and 2. Updating the network parameters using the derivatives [15]. The partial derivatives are often referred to as gradients and will ultimately be expressed in terms of errors and activations. The algorithm used for step 1 is usually backpropagation, Step 2 is commonly done using stochastic gradient descent, or a similar method.

Intuitively the backpropagation algorithm computes the error contribution of each Neuron in a NN to some loss function L. A Neuron's contribution to L depends on its corresponding weights and biases. Hence, what we want to compute is inherently $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ for all weights w and biases b in the network and change the weights and biases some amount Δw and Δb such that when the forward pass is performed again to obtain L', L' will be less than L [15].

In Section 2.4.3.1, we defined the activation of a neuron as a_j^l, per Equation (4). It is, however, less algebraically tiresome to use the weighted input of each neuron z_j^l (presented in Equation (8)) instead of the activation when computing said errors.

$$z_j^l = \left(\sum_{k=0}^{n} \left(w_{j\,k}^l a_k^{l-1} \right) + b_j^l \right) \tag{8}$$

Hence, we start by defining the error of neuron *j* in layer *l*, for *z*. as in Equation (9). δ_j^l measures how much the weighted input z_j^l affects the final loss of the network.

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} \tag{9}$$

Starting from the back of the network δ_j^L expresses how much the activation for each Neuron in the output layer contributes to the loss *L,* by Equation (10).

$$\delta_j^L = \frac{\partial L}{\partial a_j^L}\sigma'(z_j^L) \tag{10}$$

The first factor of Equation (10) measures how *L* changes depending on the activation of the j^{th} neuron. The second factor takes the activation function of the output layer into consideration and tells us how fast activation function is changing at computing the error of each neuron.

After acquiring the errors in the last layer, it is possible to propagate the error backward in the network. The propagation is done using Equation (11), which shows how the error for an arbitrary Neuron δ_j^l can be computed in terms of the error from the Neuron in the layer before.

$$\delta_j^l = \sum_{k=0}^{n} w_{kj}^{l+1}\,\delta_k^{l+1}\sigma'(z_j^l) \tag{11}$$

It is possible to repetitively apply Equation (11) and compute $\delta_j^{l-1}, \delta_j^{l-2}, \dots$, all the way back through the network, simply by using the error from the previously calculated layer.

As was referred to earlier in the section, the errors are only part of what is required to update the network parameters. We are ultimately interested in finding out how the weights and biases of the network affect the loss function. The weights effect on the loss of the network is given in Equation (12).

$$\frac{\partial L}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{12}$$

As can be seen in Figure 7 in Section 2.4.3.1, one can think of the biases as parameters whose input is always equal to one. Due to the absence of interaction between the biases and the input to the network, the erroneous contribution from the biases is unaffected by the input. Equation (13) gives the biases contributions to the error.

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \tag{13}$$

After the backpropagation algorithm has calculated the gradients, the parameter update is done by subtracting a small fraction of the corresponding gradients from each weight and bias as in Equation (14) and (15). This is the stochastic gradient method, mentioned earlier in the section.

$$w_{jk}^{l} = w_{jk}^{l} - \Delta w_{jk}^{l} = w_{jk}^{l} - \eta \frac{\partial L}{\partial w_{jk}^{l}} \tag{14}$$

$$b_{j}^{l} = b_{j}^{l} - \Delta b_{j}^{l} = b_{j}^{l} - \eta \frac{\partial L}{\partial b_{j}^{l}} \tag{15}$$

η is referred to as the learning rate of the algorithm and is usually a fixed constant; albeit variations exist, such as decaying learning rate [15].

2.4.3.3 A numerical example for training a Neural Network

Understanding backpropagation is non-trivial, especially when one has only been presented with the abstract algorithm. Hence, this section will give a numerical example of the backpropagation algorithm for a small MLP under the network seen in Figure 9.

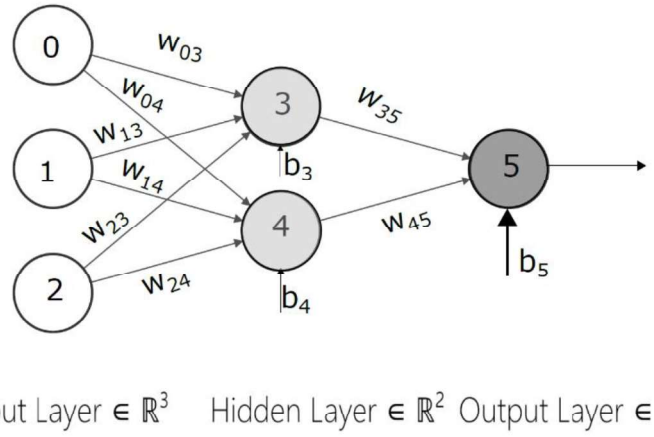

Figure 9: Shows an MLP with one hidden layer to be used in the numerical example of the backpropagation algorithm.

The task at hand is that of binary classification, hence the output layer of the network (denoted as 5) will have a single Neuron with a sigmoid function, as was presented in Equation (5). As loss function, we use $L = \frac{1}{2} \sum (y^{true} - y^{pred})^2$, which is the sum of squared errors. Furthermore, we will assume that the activation function for the hidden units (denoted by 3 and 4 respectively) is also the sigmoid function. Finally, we use a learning rate $\eta = 0.1$ to update our parameters.

The weights and biases of the MLP are randomly initiated to values in the interval [-0.5, 0.5], per Table 4. We have as input vector $x = [1\ 0\ 1]^T$ and the label of x is $y^{true}=1$.

Parameter	w_{03}	w_{04}	w_{13}	w_{14}	w_{23}	w_{24}	w_{35}	w_{45}	b_3	b_4	b_5
Value	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

Table 4: Initial values for the parameters of the MLP presented in Figure 9.

The forward pass of the MLP gives rise to values presented in Table 5, and in turn the prediction $y^{pred} = 0.474$ and the loss $L = \frac{1}{2}(1 - 0.474)^2 = 0.1383$.

Computing the backward pass of the MLP involves taking the derivative of the loss function as well as the sigmoid function, these derivatives are presented in Equation (16) and (17) respectively.

Neuron	Weighted Input	Activation
3	(0.2)(1) + (0.4)(0) + (-0.5)(1) + (-0.4) = -0.7	σ(-0.7) = 0.332
4	(-0.3)(1) + (0.1)(0) + (0.2)(1) + (0.2) = 0.1	σ(0.1) = 0.525
5	(-0.3)(0.332) + (-0.2)(0.525) + 0.1 = -0.105	σ(-0.105) = 0.474

Table 5: The weighted inputs and activation for the hidden Neurons as well as the output Neuron in the MLP depicted in Figure 9.

$$\frac{\partial L}{\partial y^{pred}} = y^{pred} - y^{true} \tag{16}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \tag{17}$$

The backward pass of the MLP starts with computing the errors, as was described in Equation (10) and (11). In total, there are three errors, one for the output node and one for each hidden node. These errors are computed and presented in Table 6. The gradients are then computed per Equation (12) and (13), yielding the result shown in Table 7.

Neuron	Error
5	(0.474 - 1)(0.474)(1 - 0.474) = -0.1311
4	(-0.2)(-0.1311)(0.525)(1 - 0.525) = 0.0065
3	(-0.3)(-0.1311)(0.332)(1 - 0.332) = 0.0087

Table 6: The errors for each Neuron in the MLP after the first backward pass of the backpropagation algorithm.

Parameters	Gradient	Updated Value
w_{03}	(0.0087)(1) = 0.0087	0.2 – (0.1)(0.0087) = 0.19913
w_{04}	(0.0065)(1) = 0.0065	-0.3 – (0.1)(0.0065) = -0.30065
w_{13}	(0.0087)(0) = 0	0.4 – (0.1)(0) =0.4
w_{14}	(0.0065)(0) = 0	-0.1 – (0.1)(0) = -0.1
w_{23}	(0.0087)(1) = 0.0087	-0.5 – (0.1)(0.0087) = -0.50087
w_{24}	(0.0065)(1) = 0.0065	0.2 – (0.1)(0.0065) = 0.19935
w_{35}	(-0.1311)(0.332) = -0.0435	-0.3 + (0.1)(0.0435) = -0.29565
w_{45}	(-0.1311)(0.525) = -0.0688	-0.2 + (0.1)(0.0688) = -0.19312
b_3	0.0087	-0.4 – (0.1)(0.0087) = -0.44087
b_4	0.0065	0.2 – (0.1)(0.0065) = 0.19935
b_5	-0.1311	0.1 + (0.1)(0.1311) = 0.11311

Table 7: The gradients, as well as the updated values, for each parameter in the MLP after one iteration of the backpropagation algorithm.

A full forward pass and backward pass has now been performed. Performing a second forward pass (with the same input and label vectors as before) yields the prediction y^{pred} = 0.478 and the loss L = 0.1359. Hence, the parameters have changed such that the forecast is closer to the true label, and the loss has decreased. The initial and final values are presented in Table 8.

	Old value	New value	Target value
y^{pred}	0.474	0.478	1
L	0.1383	0.1359	0

Table 8: The predicted value and the loss before and after one iteration of the backpropagation algorithm has been performed.

2.4.4 Tuning the performance of a Neural Network

When discussing the performance of a Neural Network configuration, two common terms are under-fitting and over-fitting. Under-fitting refers to a configuration that is unable to minimize the cost function sufficiently during training. The resulting predictions, both during training and testing, will likely be weak. Over-fitting occurs when the NN configurations are too flexible. The cost function is minimized during training, but as the Network is evaluated on unseen data, the error is significantly larger. That is, if a training sample is provided to the configuration, the output will likely be good, but if a previously not seen sample is provided, the prediction will be poor [15].

There are multiple ways in which it is possible to tune the performance of a NN configuration, and this section discusses the importance of appropriate data, the shape of a NN model, as well as different regularization techniques used during the actual training of a configuration.

2.4.4.1 Normalizing data

Commonly, the raw input data is unsuited for many NN configurations, and transforming the data can greatly improve performance.

Our standard pre-processing techniques is the normalization of data. Normalizing is often done such that the training data is rescaled to the range [0,1] or [-1,1], in accordance with what is seemed best suited for the used activation functions. This is done because common activation functions used in most NN configurations greatly benefit from this.

2.4.4.2 Learning Rate

The learning rate is one of, if not the most important hyperparameter. If this is too large or too small, a network may learn very poorly, very slowly, or not at all. Most commonly practiced values for the learning rate are in the range of 0.01 to 1e-6, though the optimal learning rate is usually data (and network architecture) specific. Some simple method is to start by trying three different learning rates – 1e-1, 1e-3, and 1e-6 – to get a rough idea of what it should be, before further tuning this. Ideally, they run models with different learning rates simultaneously to save time.

The usual approach to selecting an appropriate learning rate is to visualize the progress of training (using Tensorboard). Pay attention to both the loss over time and the ratio of update magnitudes to parameter magnitudes (a ratio of approximately 1:1000 is a good place to start). For training neural networks in a distributed manner, may need a different (higher) learning rate compared to training the same network on a single machine.

2.4.4.3 Regularization

Regularization methods can help to avoid overfitting during training. Overfitting occurs when the network predicts the training set very well but makes weak predictions on test/validation data. One way to think about overfitting is that the network memorizes the training data (instead of learning the general relationships in it).

Common types of regularization include:

- L1 and L2 regularization penalizes large network weights and avoids weights becoming too large. L2 regularization is commonly used in practice. However, note that if the L1 or L2 regularization coefficients are too high, they may over-penalize the network, and stop it from learning.
- Dropout is a frequently used regularization method can be very effective. Dropout is most commonly used with a dropout rate of 0.5. The Dropout technique randomly masks out hidden unit activations during training, which prevents co-adaption of hidden units.
- Restricting the total number of network size (i.e., limit the number of layers and size of each layer)
- Early stopping which halts loss function optimization before ultimately converging to the lowest possible function value. Early stopping as a regularization technique is similar to an L2 weight norm penalty.

2.4.4.4 Minibatch Size

A minibatch means the number of examples used at a time when computing gradients and parameter updates. In practice (for all but the smallest data sets), it is standard to break your data set up into several mini-batches. The ideal minibatch size will vary. For example, a minibatch size of 10 is frequently too small for GPUs but can work on CPUs. A minibatch size of 1 will allow a network to train but will not reap the benefits of parallelism. 32 may be a sensible starting point to try, with mini-batches in the range of 16-128 (sometimes smaller or larger, depending on the application and type of network) being common.

2.4.4.5 Optimization Algorithm

Having defined NN architectures and the loss function we wish to optimize, we must specify which gradient-based algorithm we used to find a local minimum of our loss function. Considering only stochastic gradient techniques as batch optimization, which requires computing the gradient across the entire dataset at each step. There are several variants of stochastic gradient techniques.

2.4.4.6 Cell units

There are different kinds of supervised learning-tasks, and even though the MLP introduced in Section 2.4.2 is a common NN configuration for many supervised learning tasks, it is not the best-suited configuration for all such tasks. Revisiting the image classification example in Section 2.4.1, an image of cat remains an image of a cat regardless of any changes made to the remaining photos of the dataset. However, two adjacent samples in speed-profile time-series are related by physical limitations concerning acceleration and deceleration, and information

about the first of the two samples gives information conveyed in the sequence of data. One such technique is Recurrent Neural Networks.

2.4.4.7 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are networks with loops. They attempt to capture the information that could be found in the sequence of data, by allowing for past information to remain in the network after the data has first been processed. One can think of RNNs as networks with memory, and Figure 10 shows how a single RNN cell is connected both to some succeeding layer and to itself. The data that is passed from the cell to itself gives rise to what is usually called the hidden state of the cell.

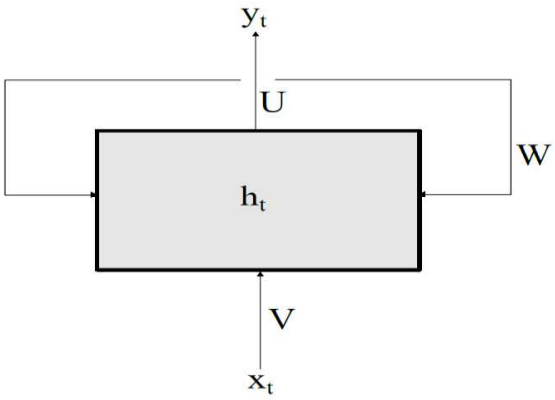

Figure 10: Shows a single RNN cell with input xt, output yt, hidden state ht, and parameter matrices V, W, and U denoted.

Equation (18) describes how the RNN cell computes its hidden state, h_t. **V**, **W,** and **U** are parameter matrices with weights and biases which are trained using Backpropagation Through Time (BPTT), analogously to the feedforward network presented in Section 2.4.3.2 but adhering to the temporal aspect of the network. The output y_t, in Equation (19), is the activation of the weighted hidden state of the RNN cell, analogously to the output layer of an MLP. Note that the activation function for the hidden state g_1 and the one which yields the output g_2 must not be the same.

$$h_t = g_1(Vx_t + Wh_{t-1}) \tag{18}$$

$$y_t = g_2(Uh_t) \tag{19}$$

The vanilla RNN and BPTT do however have a shortcoming: As the time span increases, long-lived dependencies from the computed gradients tend to either vanish or explode. This behavior occurs because the gradients of the hidden state are propagated by multiplication, causing the gradients to exponentially go towards either infinity (explode) or zero (vanish). Hence, conventional RNNs are deemed impractical for predictions spanning over a large time period.

2.4.4.8 Long Short-Term Memory (LSTM)

Long Short-Term Memory is a technique proposed by Hochreiter and Schmidhuber as a response to the issues with conventional RNN's inability to handle dependencies over large time spans. A NN utilizing LSTM cells is also recurrent, but the network circumvents the issue with exploding/vanishing gradients by using a constant error carrousel (CEC). The CEC keeps the so-called error flow constant through the cell. Hochreiter and Schmidhuber achieved this by linearly incorporating the state of the cell in the next step in time, instead of multiplying it as in BPTT.

LSTM cells come in multiple shapes and forms; hence, it is difficult to give a general description of the technique Figure 11 depicts the internals of an LSTM cell as was first presented by Hochreiter and Schmidhuber. In the cell below, the memory content s_c is protected by a multiplicative input gate which is trained to not allow for irrelevant inputs to perturbation the content; likewise, a similar output gate protects other units from perturbation, should the cell store irrelevant content.

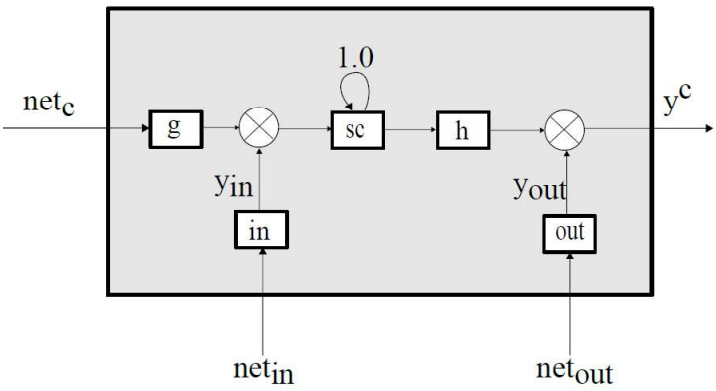

Figure 11: Shows architecture of an LSTM cell as described by Hochreiter & Schmidhuber [28].

The inputs to the LSTM cell are up to the user to specify. Hochreiter and Schmidhuber said in their original article that: "input units, gate units, memory cells, or even conventional hidden units […] may convey useful information about the current state of the net". Hence, the inputs to the LSTM cell are denoted simply as net, with a subscript to distinguish between them.

The internal state of the cell is calculated following Equation (20), which in turn gives rise to the output of the cell, presented in Equation (21).

$$s_c(t) = s_c(t-1) + y^{in}(t)g\big(net_c(t)\big) \tag{20}$$

$$y^c(t) = y^{out}(t)h(s_c(t)) \tag{21}$$

Even though there are multiple implementations of the LSTM cell, they all incorporate the CEC.

2.4.5 Neural Network Model for predicting sequences

Among several NN configurations in this thesis Sequence to Sequence model is implemented. And this section provides a thorough description of Sequence to Sequence model which is suitable for performing time-series predictions.

2.4.5.1 Sequence-to-Sequence Model

Sequence-to-Sequence (Seq2Seq) model are used for everything from chatbots to speech-to-text to dialog systems to Question and Answer to image captioning. Seq2Seq models are a class of RNNs. They allow mapping input sequences of different lengths, which is not possible with one RNN as well as Seq2Seq model's sequence preserves the order of the inputs, which is not the case with basic Neural Networks. There's certainly no good way to represent the concept of time and of things changing over time, so the Seq2Seq models allow us to process information that has a time, or an order of time (i.e., Velocity profile), element attached to it. They allow us to preserve information that could not be by a Normal Neural Network.

In simple terms, a Seq2Seq model consists of two separate RNNs, the encoder, and the decoder. An encoder takes the information as input in multiple time steps and decodes the input sequence into a context vector [29]. The decoder receives that hidden state and decodes it into the desired output sequence. Seq2Seq Model works in two phases *Training* and *Generating / Inference*.

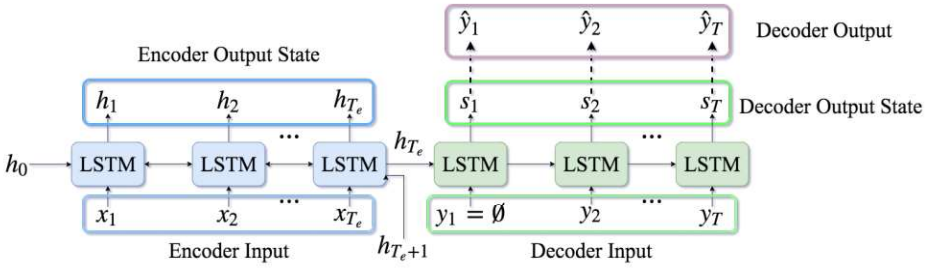

Figure 12: Shows a simple Seq2Seq Training model [29].

In Figure 12 The blue boxes correspond to the encoder part, which has T_e units. The green boxes refer to the decoder, which has a T number of units.

The encoder and decoder are usually implemented by Recurrent Neural Networks (RNN) using Long Short-Term Memory (LSTM) cell. The encoder takes a sequence of length Te inputs, $X = \{x_1, x_2, \dots, x_{Te}\}$, where $x_t \in A = \{1, \dots, |A|\}$ is a single input coming from a range of possible inputs (A) and generates the output state h_t. In addition, each encoder receives the previous encoder's hidden state, h_{t-1}. The decoder, on the other hand, takes the last state from the encoder, i.e., h_{Te} and starts generating an output of size $T < T_e$, $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_T\}$, based on the current size of the decoder s_t and the ground-truth output y_t. The decoder could also take as input an additional context vector c_t, which encodes the context to be used while generating the output. The RNN learns a recursive function to compute s_t and outputs the distribution over the next output:

$$h_{t'} = \Phi_\theta(x_{t'}, h_t) \tag{22}$$

$$s_{t'} = \Phi_\theta(y_t, s_t /h_{T_e}, c_t) \tag{23}$$

$$\hat{y}_{t'} = \pi_\theta(y|\hat{y}_t, s_{t'}) \tag{24}$$

Where $t' = t+1$, θ denotes the parameters of the model, and the function for π_θ and Φ_θ, depends on the type of RNN. A simple RNN would use a sigmoid function for Φ and a softmax function for π:

$$s_{t'} = \sigma(W_1 y_t + W_2 s_t + W_3 c_t) \tag{25}$$

$$o_{t'} = softmax(W_4 s_{t'} + W_5 c_t) \tag{26}$$

Where, o_t is the output distribution of size $|A|$ and the output $\hat{y}_t$, is selected from this distribution. W_1, W_2, W_3,W_4, and W_5 are matrices of learnable parameters of sizes $W_{1,2,3} \in R^{d \times d}$ and $W_{4,5} \in R^{d \times |A|}$, where d is the size of the input representation. The input to the first decoder is a special input indicating the beginning of a sequence, denoted by $y_0 = \phi$ and the first forward hidden state h_0 and the last backward hidden state h_{Te+1} for the encoder are set to a zero vector. Moreover, the first hidden state for decoder s_0 is set to the output that is received from the last encoding state, i.e., h_{Te}.

The most widely used method to train the decoder for sequence generation is called the teacher forcing algorithm, which minimizes the maximum-likelihood loss at each decoding step. Let us define $y = \{y_1, y_2, \ldots, y_T\}$ as the ground-truth output sequence for a given input sequence X. The maximum-likelihood training objective is the minimization of the following cross-entropy (CE) loss:

$$\mathcal{L}_{CE} = -\sum_{t=1}^{T} log\ \pi_\theta(y_t|y_{t-1}, s_t, c_{t-1}, X) \tag{27}$$

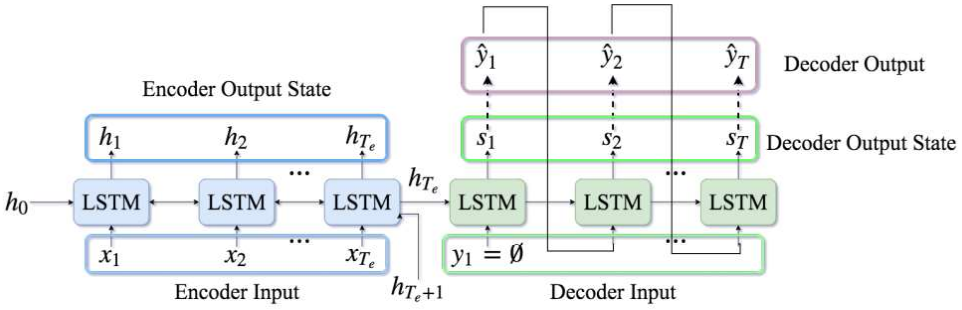

Figure 13: Shows a simple Seq2Seq generating model.

Once the model is trained with the above objective, the model generates an entire sequence as follows: Let $\hat{y}_t$, denotes the action (output) taken by the model at time t. Then, the next action is generated by:

$$\hat{y}_{t'} = \arg_y max\ \pi_\theta(y|\hat{y}_t, s_{t'}) \tag{28}$$

3 Markov Chains

As explained in Section 2.3.1 to implement the Markov Chains model, we require states and transition probability of the same. In our case, our velocities are considered as states for Markov Chains. To implement the Markov Chains, in the thesis, we have considered only one driving cycle to generate states and Transition Probability Matrix. Measured data in driving cycle is at discrete time.

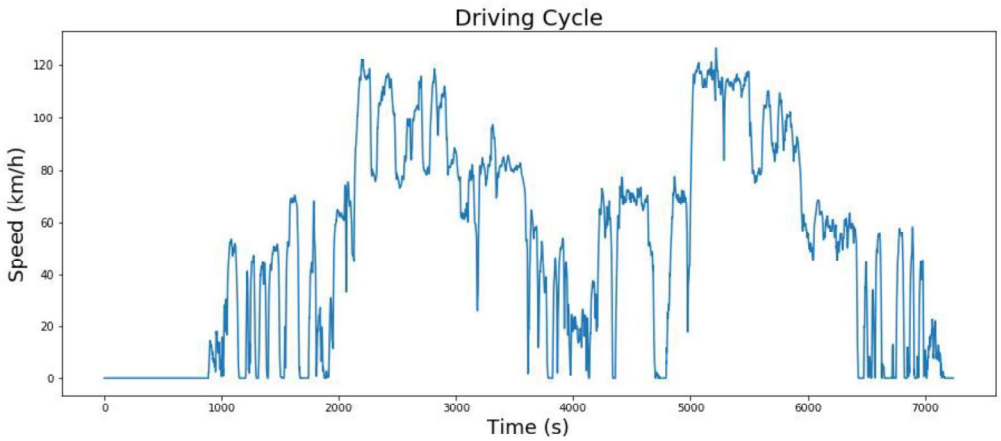

Figure 14: Shows the speed profile being used for implementing Markov Chains.

3.1 Pre-processing Data

Aforementioned speed profile was measured for 7241 seconds. And the measured value of speed is at each 2 Hz (hence, we have 7241x2 = 14482-speed values). As we can see from Figure 14, the speed profile speed at the beginning and the relatively small part of the end remains 0 km/h. That part we can remove as we can consider that the vehicle was at idle (not driving). After removing the "not driving" data driving cycle looks as shown in Figure 16.

As the speed profile is having speed values in float; hence, all speed values are being converted into a speed category (i.e., 22.4 km/h is converted to 22 km/h). And as a result of that after implementing the Markov Chains model, the forecast speed will also be in the speed category. The step between two-speed categories is 1 km/h, resulting in 127 possible speed categories that the process can adopt.