2.3 Machine Learning

The term machine learning was coined by Samuel in the 1950's. 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 category of an algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an output while 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 technique is given in **Section 2.3.1.**

2.3.1 Supervised learning

In supervised learning, algorithms learn from labeled data. 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 can be divided into 2 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 previous 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 an arbitrary data (given in **Table 2.3.1)**. Regression models have various types of model for 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 2.3.1: 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. In the **Figure** *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.

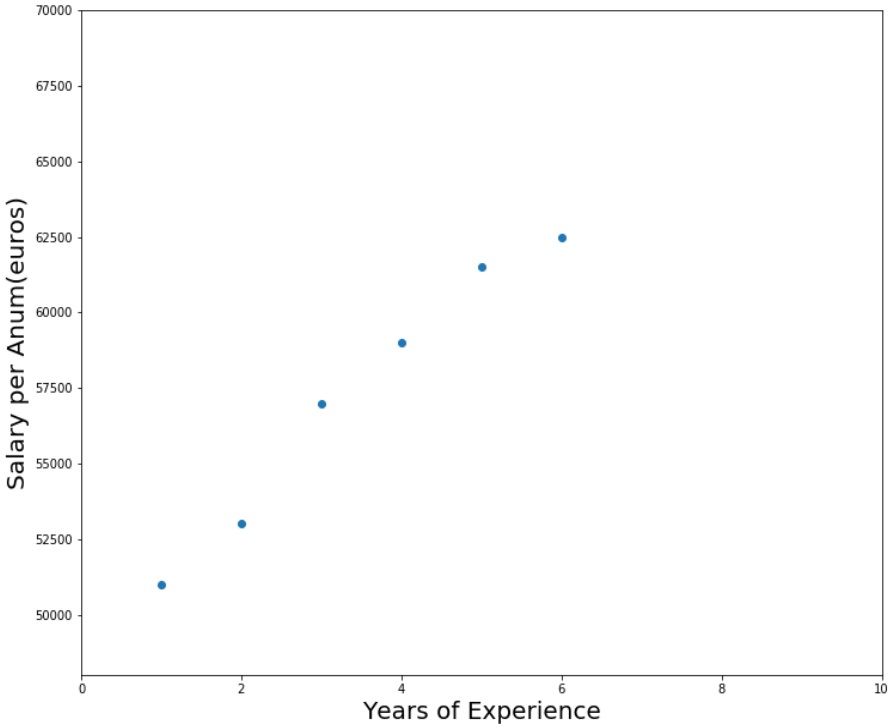

Figure 23.:

When training the model- it fits the best line to predict the value of *y* 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 β and θ values, we get the best fit line. So, when we are finally using 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 best fit line, we need to update β and θ values, to do so we will use cost function.

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

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

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

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

To update β and θ values in order to reduce Cost function (minimizing RMSE value) and achieving the best fit line the model uses Gradient Descent(**Equation second)**. 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

showed in **Figure 233432423.** And from **Figure 2323** we can obtain the expected approximately salary after having 8 years of experience would be somewhere around 68250 Euros.

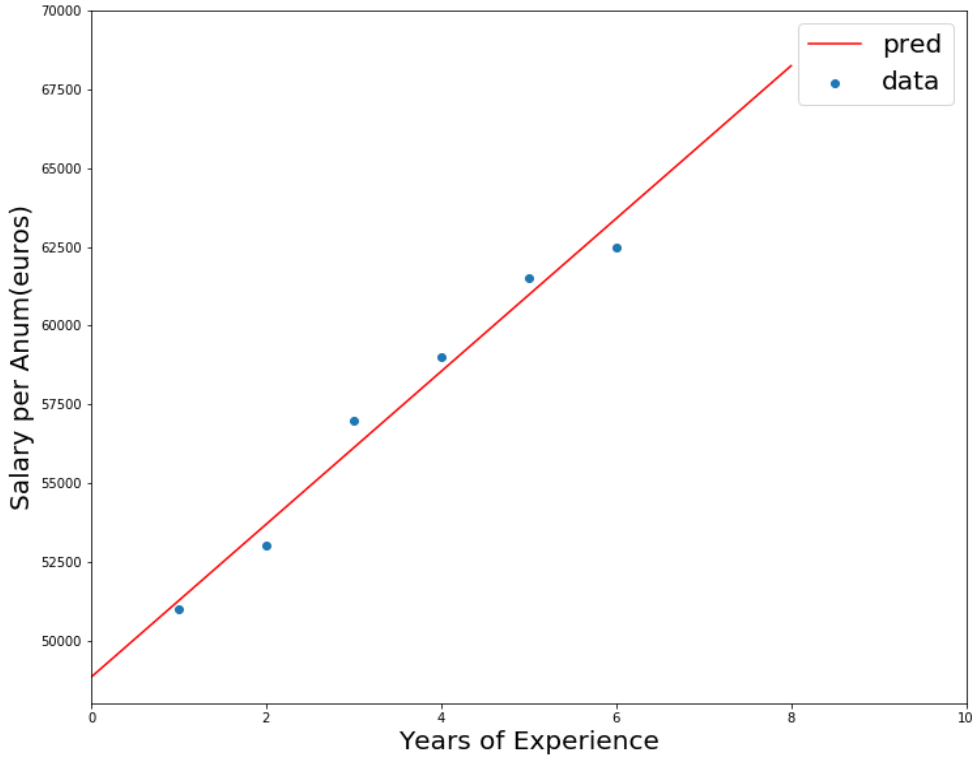

Figure 234324:

2.3.1.1 Time-series prediction as a supervised learning problem

Time-series do not have an obvious input/output-mapping, like the labeled pictures with corresponding target values in **Section 2.3.1,** but it is possible to transform a time-series into a form apt for supervised learning.

Table 2.2 shows the first 5 steps of the speed profile time-series depicted in **Figure 2.1. Table 2.3** shows the result after shifting all values in the original time-series one step in time. A mapping is created where the values in the original time-series are used as inputs *x* and the values in the shifted time-series are used as outputs *y*.

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

Table 2.2: The first five values from the speed-profile time-series shown in **Figure 2.1.**

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

Table 2.3: The values in **Table 2.2** as a supervised learning problem.

The transformation preserves order and can be extended to both multivariate time-series and for multi-step predictions (i.e. predicting multiple steps in time). Note that there is no input value for the first target value in **Table 2.3.** Neither is there any target value for the last input value in the table. During training, both these rows are commonly removed. The result of the transformation is given in **Table 2.4.**

x	Y
0.00	3.75
3.75	5.25
5.25	4.00
4.00	8.50

Table 2.4: The resulting table after transforming the values of the time-series in **Table 2.2** into a dataset apt for supervised learning.

2.3.2 Neural Network (NN)

The foundation for Neural Network was laid when McCulloch and Pitts modelled a simple Neural Network with electrical circuits, in an attempt to describe how biological neurons might work. Today, there are multiple variations of Neural Networks, but "Vanilla Neural Network "most commonly refers to the multilayer perceptron.

Multilayer Perceptrons (MLP) are commonly used for classification. The MLP is a fully connected, feedforward neural network with at least three layers. Being a fully connected NN implies 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. **Figure 2.1** depicts an MLP with four inputs, two hidden layers, and one output.

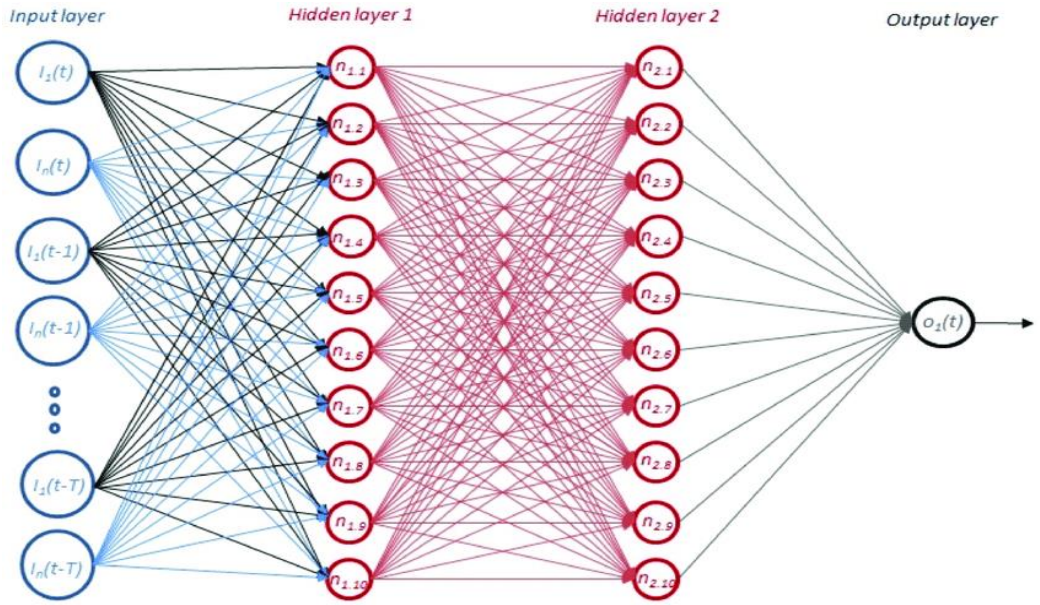

Figure 2.1: A multilayer perceptron with four inputs, two hidden layers (with ten neurons in each layer) and a single output neuron in its output layer.

Each line in **Figure 2.1** 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.

2.3.3 Training a Neural Network

All NNs must be trained before they are useful. At first, the weights and biases in a network are randomly assigned – usually in a small interval around zero – and the predictions of the network will be nothing but random. Training a NN is an iterative process where each iteration constitutes of two steps: 1. A forward pass of data to retrieve a prediction and 2. A backward pass to update the parameters of the network.

2.3.3.1 Acquiring a prediction from a Neural Network

The forward pass of MLP yields a prediction. With the exception of the Neurons in the input layer, each neuron in an MLP applies a nonlinear activation function *g* to a linear combination of the data provided by the previous layer, as is visualised in **Figure 2.2**

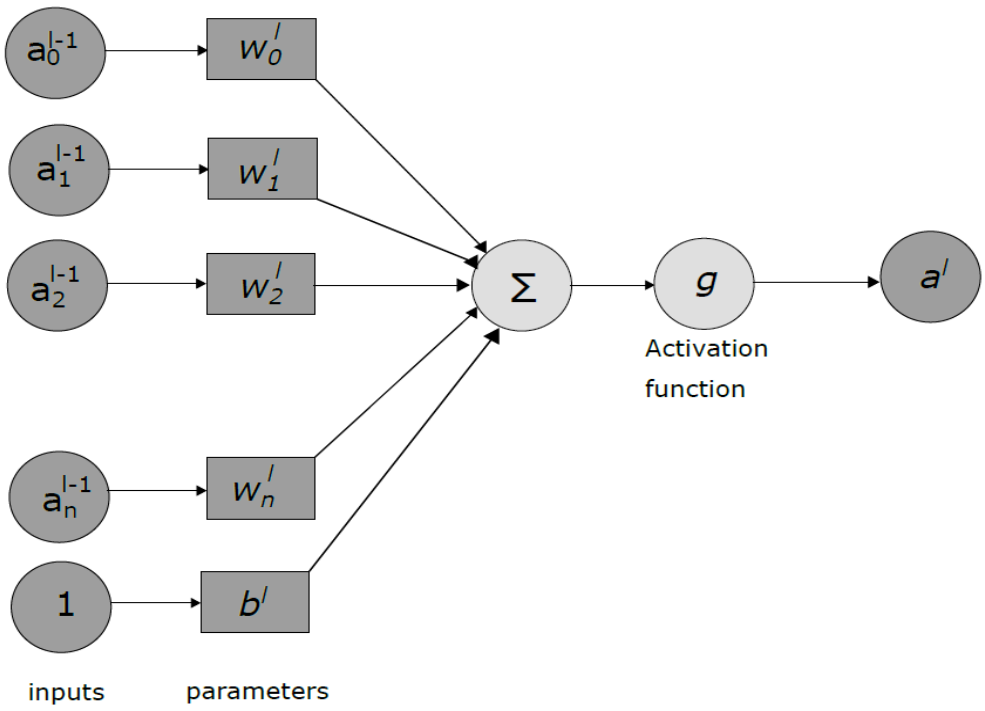

Figure 2.2 The intervals of a Neuron in a multilayer perceptron.

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*, in accordance to **Equation 2.1**. 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 denotes the bias of the j^{th} Neuron in the l^{th} layer. In **Figure 2.2** we assume that layer *l* contains only one Neuron, why we may omit the double indices.

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

It is possible that the activation function *g* differs between layers in an MLP. As was mentioned in ***section of Supervised learning,*** the softmax function is commonly used as the activation function for the output layer, in order to retrieve a vector which can represent a probability distribution over some classes. However, it is more common to use the sigmoid function (**Equation 2.2**), the hyperbolic tangent function (**Equation 2.3**) or the rectified linear unit (**Equation 2.4**) as activation functions for the Neurons in the hidden layers.

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

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

$$ReLU(t) = \max(0,t)$$

2.3.3.2 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* of the network. The process constitutes of two steps: 1. Computing partial derivatives and 2. Updating the network parameters using the derivatives. 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, an algorithm discovered by multiple researches independent of one another over a period spanning the late 1960's to mid-1980's. 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 Δ*w* and Δ*b* such that L' < L when the forward pass is performed again to acquire L'.

In **Section 2.3.2.1** we defined the activation of a neuron as a_j^l, in accordance to Equation 2.2. It is, however, less algebraically cumbersome to use the weighted input of each Neuron z_j^l (presented in Equation 2.6) instead of the activation when computing said errors.

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

Hence, we start by defining the error of Neuron *j* in layer *l*, with respect to *z*. as in Equation 2.7. δ_j^l measures how much the weighted input z_j^l affects the final loss of the network.

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

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,* in accordance to **Equation 2.8.**

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

The first factor of **Equation 2.8** 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. This is only for the last layer of the network and the motivation for using the activation is given ion **Equation A.1**

After acquiring the errors in the last layer, it is possible to propagate the error backwards in the network. The propagation is done using **Equation 2.9,** 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. The derivation of the equation is given in **Equation A.2.**

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

It is possible to repetitively apply **Equation 2.9** 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 affects the loss function. The weights effect on the loss of the network is given in **Equation 2.10.**

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

As can be seen in **Figure 2.5** in **Section 2.4.1**, one can think of the biases as parameters whose input is always equal to one. Due to absence of interaction between the biases and the input to the network, the erroneous contribution from the biases are unaffected by the input. **Equation 2.11** gives the biases contributions to the error.

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

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 2.212 and 2.13.** This is the stochastic gradient method, mentioned earlier in the section.

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

$$b_j^l = b_j^l - \Delta b_j^l = b_j^l - \eta \frac{\partial L}{\partial b_j^l}$$

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 [16].

2.3.3.3 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 in accordance to the network seen in **Figure 2.6.**

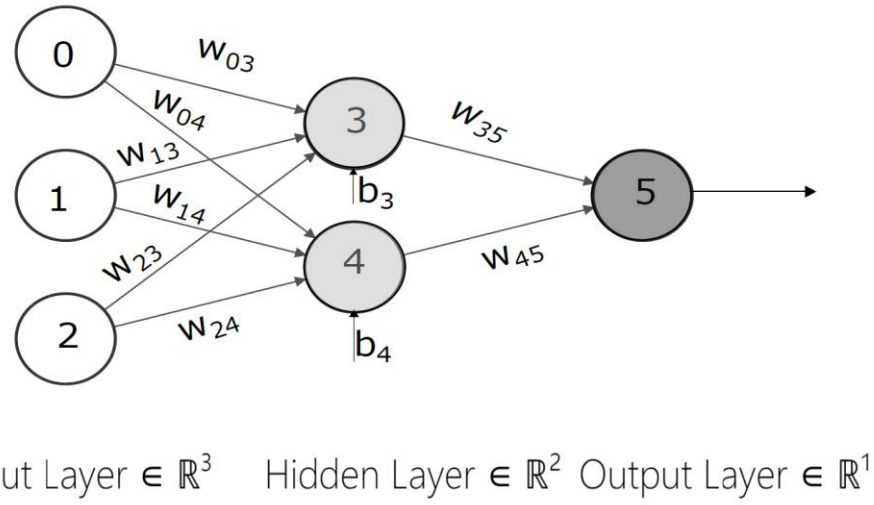

Figure 2.6: An MLP with one hidden layer to be used in the numerical example of the backpropagation algorithm.

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 **Equation2.3.** As loss function we use $L = ½ \sum(y^{true}-y^{pred})^2$, 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 digmoid 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], in accordance to **Table 2.5.** 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 2.5: Initial values for the parameters of the MLP presented in **Figure 2.6.**

The forward pass of the MLP gives rise to values presented in **Table 2.6,** and in turn the prediction $y^{pred} = 0.474$ and the loss $L = ½ (1 – 0.474)^2 = 0.1383$.

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 2.14** and **2.15** respectively.

Neuron	**Weighted Input**	**Activation**
3	0.2 + 0 - 0.5 - 0.4 = -0.7	σ(-0.7) = 0.332
4	-0.3 + 0 + 0.2 + 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 2.6: The weighted inputs and activation for the hidden Neurons as well as the output Neuron in the MLP depicted in **Figure 2.6.**

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

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

The backward pass of the MLP starts with computing the errors, as was described in **Equation 2.8 and 2.9.** 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 2.7.** The gradients are then computed in accordance to **Equation 2.10 and 2.11, yielding** the result presented in **Table 2.8.**

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 2.7: 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 2.8: 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 th loss L = 0.1359. Hence, the parameters have changed such that the prediction is closer to the true label and the loss has decreased. The initial and final values are presented in **Table 2.9.**

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

Table 2.9: The predicted value and the loss before and after one iteration of the backpropagation algorithm has been performed.

2.3.3.4 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 poor. Over-fitting occurs when the NN configurations is too flexible. The cost function is properly 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.

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.

2.3.3.4.1 Normalising data

It is common that the raw input data is unsuited for many NN configurations and transforming the data can greatly improve performance.

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

2.3.3.4.2 Learning Rate

The learning rate is one of, if not the most important hyperparameter. If this is too large or too small, network may learn very poorly, very slowly, or not at all. Typical values for the learning rate are in the range of 0.1 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. 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 (frequently higher) learning rate compared to training the same network on a single machine.

2.3.3.4.3 Regularization

Regularization methods can help to avoid overfitting during training. Overfitting occurs when the network predicts the training set very well but makes poor predictions on data the network has never seen. 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. Some level of 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. Common values for l2 regularization are 1e-3 to 1e-6.
- Dropout, is a frequently used regularization method can be very effective. Dropout is most commonly used with a dropout rate of 0.5.
- Dropconnect (conceptually similar to dropout, but used much less frequently)
- Restricting the total number of network size (i.e., limit the number of layers and size of each layer)
- Early stopping

2.3.3.4.4 Minibatch Size

A minibatch refers to 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 minibatches. 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 minibatches in the range of 16-128 (sometimes smaller or larger, depending on the application and type of network) being common.

2.3.4 Cell units

There are different kinds of supervised learning-tasks and even though the MLP introduced in **Section 2.3.2(Neural Networks)** 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 2.3.1** (Supervised learning), 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.

2.3.4.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are networks with loops. They attempt to capture the information that could possible be found in 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 2.8** 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.

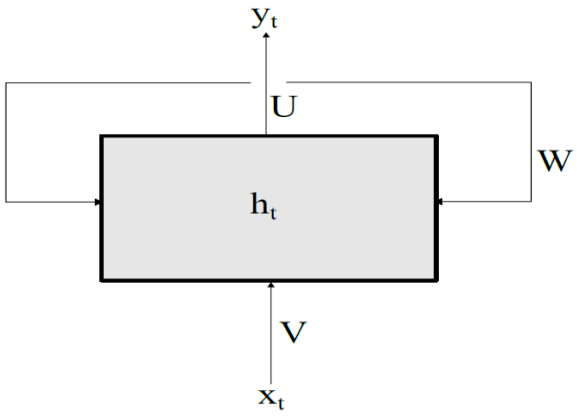

Figure 2.8: A single RNN cell with input x_t, output y_t, hidden state h_t, and parameter matrices **V**, **W** and **U** denoted.

Equation 2.16 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 2.4.2** (Updating the parameters of a NN) but adhering to the temporal aspect of the network. The output y_t, in **Equation 2.17,** 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.

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

$$y_t = g_2(Uh_t)$$

The vanilla RNN and BPTT does however have a shortcoming: As the timespan 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.

2.3.4.2 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 2.9** depicts the internals of an LSTM cell as it 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 perturbate the content, Likewise, a similar output gate protects other units from perturbation, should the cell store irrelevant content.

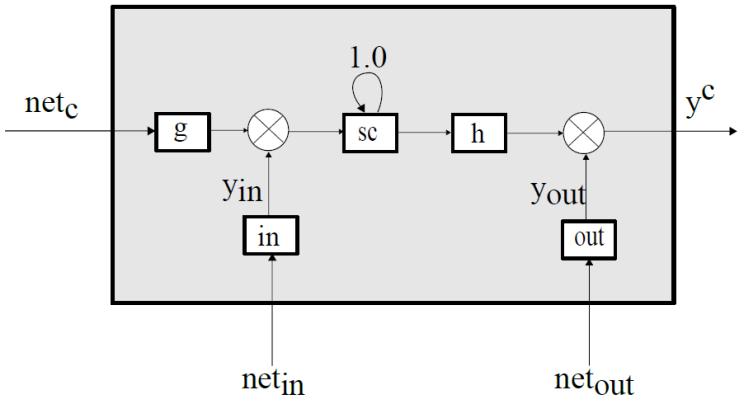

Figure 2.9: The architecture of an LSTM cell as described by Hochreiter and Schmidhuber.

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 in accordance to **Equation 2.18,** which in turn gives rise to the output of the cell, presented in **Equation2.19.**

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

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

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

2.3.5 Different models for predicting sequences

A NN configuration is more than its cell units, the cell units need to be structured as a model. This section provides a through description of two models which are suitable for performing time-series predictions.

2.3.5.1 One to many Recurrent Neural Network

The first model is a one to many model, visualised in **Figure 2.11.** At each step-in time the model takes a single value as input and produces a sequence as output. If the input to the model at time *t* is the velocity of a vehicle v_t, the output of the model is a vector length *N* where *N* denotes "the number of future speed samples to predict". If *N* = 5 the model input will be x = [v_t] and the model output will be $y^{pred} = [v_{t+1}^{pred} \ v_{t+2}^{pred} \ v_{t+3}^{pred} \ v_{t+4}^{pred} \ v_{t+5}^{pred}]$.

The motivation behind the model is that relevant information about the history of samples could be learned by the time dependant cell units.

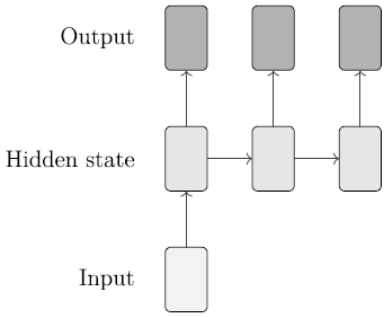

Figure 2.11 The principal layout behind a one to many model.

2.3.5.2 Many to many Recurrent Neural Network

The one to many model in **Section 2.6.1** attempts to predict some number of upcoming samples using only one sample as input. However, previous (v_{t-1}, v_{t-2},…, v_0) could possibly convey some relevant information that is not already captured in the hidden state of the network. In other words, it might be beneficial to provide the network with a sequence as input. The many to many model takes a fixed number of inputs *M* and produces a fixed number of outputs *N*.

If *M* = *N* = 5, the input to the model at time *t* will be a vector x =[$v_{t-4} \ v_{t-3} \ v_{t-2} \ v_{t-1} \ v_t$] and the output of the model will be a vector $y^{pred} = [v_{t+1}^{pred} \ v_{t+2}^{pred} \ v_{t+3}^{pred} \ v_{t+4}^{pred} \ v_{t+5}^{pred}]$.

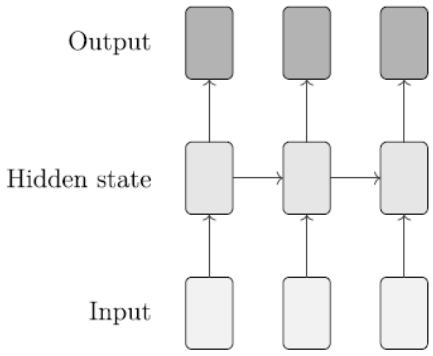

Figure 2.12: The principal layout behind a many to many model.