Designing a deep-learning network for traffic density and volume prediction

Simon Sjögren

Bachelor's thesis
2019:K13

LUND UNIVERSITY

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

Designing a deep-learning network for traffic density and volume prediction

Simon Sjögren

Lund university Faculty of science

Abstract

The goal of this thesis is to construct and train two different of neural networks whose purpose is to predict traffic volume into the future where each prediction is based on historical data. The two types of neural networks constructed for this project is the convolutional neural network (CNN) and the Long Short-Term Memory (LSTM) neural network. For the training data, two different data-sets were used. Each data-set is split with respect to time-steps where the time-interval between each time-step is uniform. The first data-set has 960 time-steps and at each time-step the dimensions of the data is 14×14 where each entry in this matrix represents a checkpoint which registered the numbers of vehicles passing this checkpoint within an interval of 30 seconds. With 960 timesteps and each timestep have an interval of 30 seconds, the total time this data-set spanned was roughly 8 hours. This road, where the data comes from have six lanes and a total of 28 checkpoints for each lane, but in order for this data to be formated as a square (which was easier to work with), a row vector and a column vector of zeros were added. The reason CNN was used for this project was to observe if the CNN could predict time-series data by finding features in the data-set. Because of the sizes of the data-set at each time-step is 14×14, the CNN reduces this input to a smaller matrix, this will reduce the number of neurons and parameters for the fully connected layers. In general, if the input data at each time-step is very big (for instance 200×200) the CNN reduces this input to something smaller so that the number of parameters in the neural network is small enough for the computer to handle. The LSTM is used because it is specifically made for time-series data and prediction. The LSTM type network compared to the CNN was also much cheaper to train and the amount of parameters was one fifth to that of the CNN using the same data-set. The 2nd data-set had a vector of size 4 for each time-step so only the LSTM type network was sufficient. As in the first data-set, each entry represents a time checkpoint. However for this data-set, each time-step had an interval of five minutes and the total number of time-steps is a little over 50000, therefore this data-set spanned several months.

The convolutional neural network was first trained on data-set 1. Training a CNN type network on time-dependant data gave poor results for both the training set and the validation set which means this model was unable to learn. Then a LSTM type network was used for the first data-set as well which resulted in over-fitting on the training data, and as a result the prediction on the validation set gave poor results as well. Despite the poor results of the validation set, this model had the potential to learn, but unfortunately the data-set had too few time-steps. The 2nd data-set was trained on another bigger LSTM type network with relative success. This training was repeated a total of three times. For the first training session, the ReLU (Rectified Linear Unit) activation function was used on the fully connected layers. The second training session used the activation function ELU (Exponential Linear Unit) on the fully connected layers and finally the third training session used the training input as training output with ReLU as the activation function. Training on different activation functions led

to the ReLU function to be superior over the ELU activation function and using the same input as desired output resulted in similar accuracy.

The end result using the second data-set (with ReLU) gave satisfactory results and some of the predictions could even predict positive and negative trends before the trends occurred in the real data. To compare all prediction sets with the real data, both the average and maximum relative error of each prediction set was calculated. The relationship between the average and the maximum relative error was observed to be linear.

Acknowledgements

I would like to thank my supervisor Alexandros Sopasakis for his much valued help and teaching during this project. Thanks to him, my knowledge in machine learning has been expanded tenfold. I am very much appreciative for him letting me finish this project in my own pace. I would also like to thank my friends and finally, my family for their continued support throughout my years in university and for the years to come.

Contents

1 Introduction: 6

2 Part 1: Machine Learning theory 8
- 2.1 Linear and nonlinear regression . . . 8
- 2.2 The Neural Network . . . 10
- 2.3 Metrics: . . . 11
  - 2.3.1 Types of metrics: . . . 14
- 2.4 The activation functions . . . 15
- 2.5 Overfitting: . . . 18
  - 2.5.1 Dropout . . . 18
  - 2.5.2 Data-Augmentation . . . 19
- 2.6 Regularization: . . . 21
- 2.7 Optimizers . . . 22
- 2.8 Fully connected: . . . 23
  - 2.8.1 Feedforward . . . 23
  - 2.8.2 Backpropagation . . . 25
- 2.9 Convolutional Neural Network . . . 29
  - 2.9.1 Feedforward . . . 29
  - 2.9.2 Backpropagation for convolutional layers . . . 33
- 2.10 Recurrent neural network: . . . 36
- 2.11 LSTM . . . 37
  - 2.11.1 Activation and cost functions for LSTM: Softmax and Cross Entropy . . . 38
  - 2.11.2 Feedforward of the LSTM: . . . 40
  - 2.11.3 Backpropagation for LSTM: . . . 42

3 Part 2: Implementation and results 45
- 3.1 Data-set 1: . . . 47
- 3.2 Data-set 2 with timestep shift: . . . 55
- 3.3 Data-set 2 without timestep shift: . . . 62

4 Part 3: Conclusion 69
- 4.1 The data-sets . . . 69
- 4.2 ELU vs ReLU . . . 69
- 4.3 Comparison between timestep shift vs no shift . . . 70
- 4.4 The neural network . . . 71
- 4.5 Judging the prediction sets . . . 71
- 4.6 Further work . . . 72

1 Introduction:

The ability to see in to the future has always been an exciting idea for mankind and this unknown have always caught the attention of the curious mind. People throughout the ages have always been attempting to predict certain events to happen in the near or distant future. One of the most famous of these was the great and mystical Nostradamus. Despite his fame, most of his, if not all predictions were poor and vague which could be applied to almost anything depending on who's interpreting. In more practical applications, predictions rely on the use of statistics. One of the roles of statistics is to generalize certain events in order to apply these to the future. The most well-known of these attempts is related to predicting the stock-market. A person who can find a pattern in the stock-market will surely become one of the most powerful individual in the world. With the recent innovation in software and hardware computational power, the neural network has revolutionized how we do statistics. Neural networks has the fascinating ability to approximate functions stemming from the real world. The most well known neural networks are the so called deep learning algorithms. A typical application of this is in image recognition such as recognizing handwritten numbers and facial recognition. Another application is in self-driving cars. Deep learning algorithms can also be applied to time-series data such as stock-market data.

In this thesis, the focus will be on traffic volume predictions. In order to predict with respect to this data, a type of deep learning method known as the LSTM will be used.
The basis for machine learning is statistics or more specifically Bayesian statistics. Bayesian statistics were first introduced in 1763 with Thomas Bayes (hence the name) which we now call for Bayes' theorem [18]. This theorem laid the groundwork for what we know today as machine learning. 49 years later, Laplace would expand the work of Bayes, further developing Bayes' Theorem.

Other important works which contributed to machine learning has been the discovery of the least squares method in 1805. This method is used in e.g regression for data fitting. The name is derived from minimizing the error squared between a calculated output vs the actual output. Least squares was discovered by Adrien-Marie Legendre 42 years after Bayes' theorem was published.

Other discoveries which have contributed to the developing of machine learning is Markov Chains in 1913, the invention of the Perceptron in 1957 [19], which is a function that returns a certain value depending on its input e.g:

$$f(x) = \begin{cases} 1, & wx + b > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

In 1951 the first neural network machine was built. This machine was built by Marvin Minsky and it was named SNARC [20]. SNARC is short for Stochastic Neural Analog Reinforcement Calculator. This machine simulated a rat which would solve a maze.

Other types of deep learning models was the Neocognitron [11] in 1980 which was proposed by Kunihiko Fukushima, this model was to recognize hand written zip codes. It took 3 days of training for the network to learn. The development of this led to the convolutional neural networks which is often used in image recognition. The Neocognitron was a very important step for deep learning. Another type of layer is the LSTM (Long Short-Term Memory) which is commonly used for predicting time-series data such as the stock market or traffic data. The LSTM was invented by Hochreiter and Schmidhuber in 1997 [26].

Machine learning is a big subset of statistics where different type of models can be used in order to predict outcome. Such models are but not limited to: Artificial neural networks, genetic algorithms and decision trees. In this thesis, the artificial neural networks will be used.

This thesis has a theory part which explains the details of how a neural network is initialized and how it is learning. This part will also include methods of optimizing the neural network to increase performance and accuracy.

Finally, at the end, the results will be presented with graphs and explanations as well as a conclusion with discussions. The results from the different types of models will be compared, one model using the convolutional type network and the other will use the Recurrent type network with focus on the Long Short-Term Memory unit. The convolutional type network will be trained on the first data-set only, the first LSTM type network will also be trained on the first data-set. The final network, which is a bigger LSTM type network will be trained on the second (and final) data-set. The final network will be trained three times, the first two times to compare two different type of activation functions, one of the two is the Rectified Linear Unit and the other is the Exponential Linear Unit, and for the third training session, the network will be trained on the same data-set, however, this time, the supervised learning will be modified.

The task for this project was to build a neural network in Python 3.6 using the Keras library which is an API (Application Programming Interface) utilizing TensorFlow as the backend, then using this network to predict time-series data for traffic volume.

2 Part 1: Machine Learning theory

2.1 Linear and nonlinear regression

In order to understand how a neural network operates, it is important to know linear regression. This is because the neural network is an expansion of the classical linear regression. Linear regression is about fitting a linear function

$$y = mx + b \tag{2}$$

to data where the relationship between the independent variable x and the dependant variable y is affine, m is the slope of the line and b is the intercept (the value of the function when $x = 0$)

$$y_i = mx_i + b + \epsilon_i \tag{3}$$

where the subscript denotes which data entry and ϵ_i is the error for this data entry with respect to the linear function.

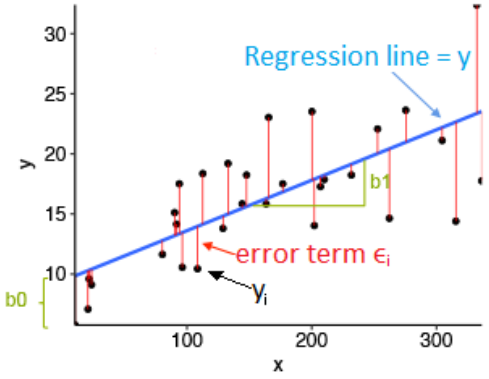


Figure 1: The X-axis is the independent variable, Y-axis is the dependant variable, b is equal to 10 and the red lines represent the error ϵ_i between the given values of y_i and the blue linear function. Note that for regression, both y_i and x_i are known [15].

Linear regression can also be used to approximate a multidimensional linear function which is represented by a plane instead of a line

$$y = m_1x_1 + m_2x_2 + b \tag{4}$$

where x_1 and x_2 are the independent variables on their own real lines and m_1 and m_2 are their respective slopes and b is the intercept (the value on the y-axis when both x_1 and x_2 are zero). This is a two-independent-input linear regression.

The downside about this type of function-fitting is that it only works for linear relationships between the independent variables and the dependant variable.

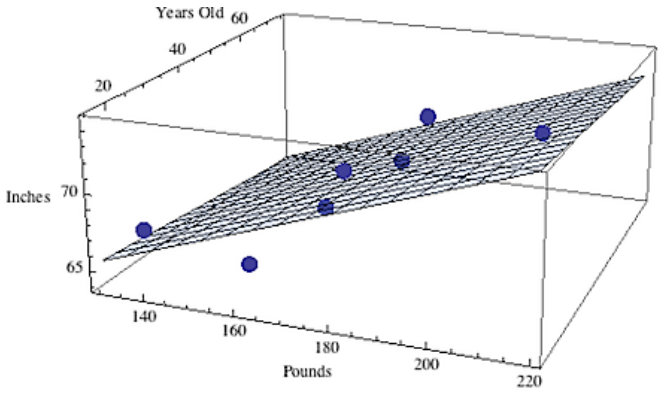


Figure 2: In this picture, the independent variables are the weight in pounds and the age of the individual. The dependant variable is the height in inches. The regression function which best fits this data is a plane. This regression function tells us that as the person ages, his height decreases, but the older people whom were chosen could also have been of same height as when they were younger. If two people of similar body-fat % and muscle mass % it is naturally to assume that a taller person weights more than a shorter person. Before drawing any conclusions, the height of a person does not increase as the person gains more weight. It is important to understand how the variables are connected in order to draw conclusions. As a child gets older and taller he will naturally gain more weight, but he can at the same time weigh more from other sources without adding height. [21]

In order to create a function which can describe the nonlinear relationships, a so called nonlinear function must be used on the linear parts of the regression. The nonlinear sigmoidal function σ is an s-shaped monotone increasing function and is usually defined as,

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

This creates non-linearity of the regression and instead of having a linear regression function we now have a nonlinear regression function which can describe nonlinear relationships between the independent variables x_i for some $i \in \mathbb{N}$ and the dependant variable y [**pp.392**, 13].

$$y = \sigma(m_1x_1 + m_2x_2 + b) \tag{6}$$

Notice how $m_1x_1 + m_2x_2$ from equation (6) is a scalar product of two vectors

$$m = \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{7}$$

This gives the nonlinear regression function the following description

$$y = \sigma(m^T x + b) \tag{8}$$

Because the scalar products of m and x is a real number, the resulting output y will also be a real number and not a vector. In order to get a multidimensional output y, the input vector x must be multiplied by a matrix instead of a vector and b must be a vector of equal dimensions to that of y. Matrix multiplication with a vector results in a vector. By replacing the vector m by a matrix W the output y will be vector instead of a scalar. The number of rows in W equals to the dimension of y.

$$y = \sigma(Wx + b) \tag{9}$$

The σ is also known as an activation function.

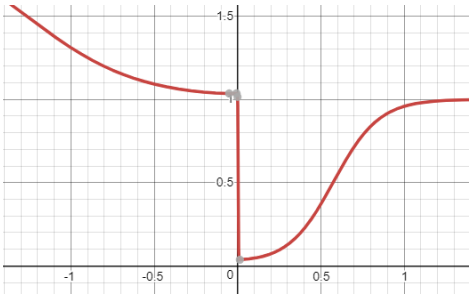


Figure 3: Example of a nonlinear function with a one dimensional input created by a sum of the nonlinear sigmoid function. All inputs for the sigmoid were of the form $mx + b$.

2.2 The Neural Network

Using the information above it is possible to set up a neural network. This neural network works similar to that of regression in the sense that the independent variables and the dependant variable are all known and these variables (both independent and dependant) are multidimensional. Because y is multidimensional from equation (9), W must be a matrix and b must be a vector and for simplicity let's ignore v as it will not be needed. Note that the vector y and the vector b must have same dimensions and the number of rows in W must be the same as entries in y and b. Because W and b are unknown they must be initiated randomly. It is common to initiate weights randomly closer to zero as having big weights usually yield poor results and weights exactly at zero will only return a linear function [**pp.397-398**, 13]. As a consequence, the output y will be far from correct. This is an approximated output of the real y. The approximated output is defined as a^L and the real output will be defined as y or in some cases Y. Similar for the input which can be defined as either x or X. The network will in essence look like this:

$$a^L(X) = \sigma(WX + b) \tag{10}$$

The neural network is essentially a multidimensional nonlinear regression function. The purpose of the activation function (which there exist many of) is to create nonlinearity. The term activation comes from the fact that a signal is sent (or activated) only when the affine function $WX+b$ is bigger than a certain tolerance. Consider the Heaviside function, this function would only return 1 if $WX + b > 0$ and return 0 otherwise.
This type of machine learning is called supervised learning; each input data x will have corresponding output data y. If we knew the weights W and bias b (as they are usually called) our approximation a^L would be equal to y and we have thus found a nonlinear-multidimensional regression function which best fits the data. In a perfect world this would be true, however, due to round-off error and ignorance of the weights and biases (W and b) this result will not be true. Instead, our output will be a^L which is an approximation of y, albeit a poor approximation. Two questions arises from these problems: how big is the error between a^L and y? And how much must we adjust W and b with respect to this error? This process of feeding the network information to then get an output a^L is called feedforward. During this process we evaluate the error between a^L and y and adjust the weights and bias W and b to reduce the error, which is called backpropagation.
But before we continue with backpropagation, it is necessary to define which error to use and which activation function to use.

2.3 Metrics:

As mentioned before, in order to compare the error and then adjust the weights and biases it is necessary to define a metric. Because K is dense in C, there exists a norm and n such that $\|f_n(x) - f(x)\|_c < \epsilon$ for some $\epsilon > 0$. The most commonly used is the Mean Squared Error.
From now on, C will be defined as the cost function and not the space of continuous functions.

$$C = (a^L(x_p) - y(x_p))^2 \in \mathbb{R}^n \tag{11}$$

This is the cost function for a particular training example x_p. This function is necessary when the network is going to learn. The metric chosen is equivalent to the Mean Squared Error(MSE) and is defined as,

$$\|a^L(x_p) - y(x_p)\|_c = \frac{1}{n}\sum_{i=1}^{n}(a_i^L(x_p) - y_i(x_p))^2 \in \mathbb{R} \tag{12}$$

This is the error between the approximation and the real function for a particular training input x_p, and n are the number of entries in the vectors $a^L(x_p)$ and $y(x_p)$. In general, the norm is a generalized metric, but in this case it will be defined as the MSE unless stated otherwise.
The loss function for a given metric of the training session is defined as

$$C_L = \frac{1}{2N}\sum_{i=1}^{N}\|a^L(x_i) - y(x_i)\|_c \in \mathbb{R} \tag{13}$$

where L denotes the loss, x is the set of training examples and N is how many training examples there are in the training-set x. The loss function is helpful to compare the error for different neural network models. To optimize a neural network for a particular data-set x, several models must be designed and trained. In the perfect world mentioned above, the cost function in equation (11) is the zero vector, but in our world, this cost function is not zero. In order for this error to be zero or close to zero, the weights W and b must be adjusted and we must know how much the adjustment is and in which direction.

The idea is then to reduce the cost function (and as a consequence the loss function) by changing the weights and bias so that the change of the cost with respect to these parameters is zero, or in mathematical terms:

$$\frac{\partial C}{\partial W} = 0, \quad \frac{\partial C}{\partial b} = 0 \tag{14}$$

Because the approximation of equation (**??**) is an element of K, the cost function is always positive with several local minima, and at least one of these minima is at or close to zero. We would like to find the smallest minima value, but this is not something we can demand.
The method to find a minima value is called stochastic gradient descent. Stochastic because the weights and biases W and b are initiated randomly.

Gradient descent means to find the gradient with respect to the weights and biases. The gradient will show us in what direction to adjust these weights and biases, it will also become smaller and smaller the closer we are to a minima value. This gradient is a tool to find the nearest local minima of the cost function. The gradient is a vector where its direction is towards the steepest part of the slope at that point, the gradient's length is the strength of the slope. Because we need the minima values, we can take the negative value of this gradient instead.

The difference between the gradient and the Jacobian is that the latter is a generalized vector derivative of the actual function whereas the gradient is an evaluated Jacobian at that point with respect to the same function (the cost function). Because we don't know what the actual function we're approximating looks like, it is impossible to use the Jacobian, therefore, the gradient of the cost function is used instead.
In other fields such as optimization, it is possible to obtain the Jacobian from a known function. This is desired because the Jacobian will help the user find a minima value. An example of this is the Rosenbrock function which is a typical optimization problem. Because this function is known, using the Jacobian is possible. Examples of methods to find a local minima is (but not limited to) the Newton method or the rank 1 Broyden update which reminds of the rank 1 weight update for backpropagation(more on this later).

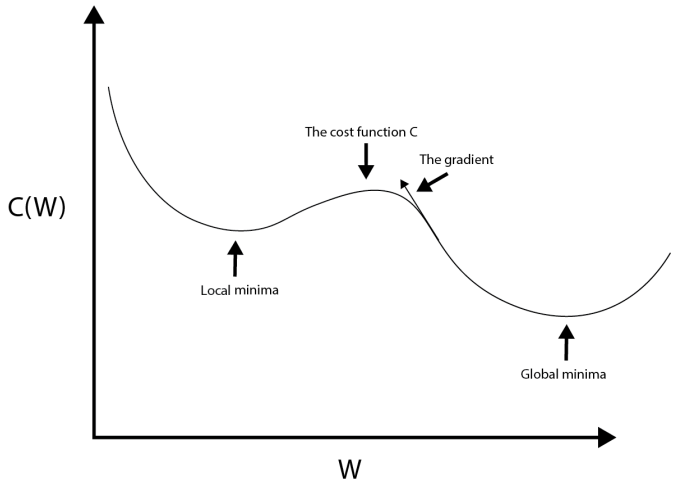


Figure 4: The cost function with respect to W in one dimension. Because the cost function is a hyper-dimensional function, the gradient will be pointing in many dimensions. The starting point of the gradient is randomly initiated. It will virtually never start close to a global minima. The gradient will move towards the nearest local minia, therefore optimizers are used to help the gradient find its way to a lower minima.

2.3.1 Types of metrics:

The metrics used for this project are the regression metrics, the ones that will be discussed are known as Mean Squared Error, Root Mean Squared Error, Mean Absolute Error. There exists other regression metrics such as the Cosine Proximity and the Mean Percentage Error but these were not used [4].

Each of these metrics will have a different loss function for each training instance. Note that the first sum $\frac{1}{2N}\sum_{p}^{N}$ is used to calculate the total loss for the training session and the second sum $\sum_{i}^{n}$ is the loss for a particular training instance.
The loss function for Mean Squared Error:

$$C_L = \frac{1}{2N}\sum_{p}^{N}\frac{1}{n}\sum_{i}^{n}(a_i^L(x_p) - y(x_p)_i)^2. \tag{15}$$

Where a^L is the approximated output and y is the desired output and N are the number of training instances.

Root Mean Squared Error:

$$C_L = \frac{1}{2N}\sum_{p}^{N}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_i^L(x_p) - y_i(x_p))^2}. \tag{16}$$

This metric represents the sample standard deviation of the differences between the prediction a^L vs desired y.

Mean Absolute Error:

$$C_L = \frac{1}{2N}\sum_{p}^{N}\frac{1}{n}\sum_{i=1}^{n}|a_i^L(x_p) - y_i(x_p)|. \tag{17}$$

The Mean Absolute Error (MAE) is more tolerant with its error because it averages the absolute differences between the prediction a^L and the desired output y linearly, this mean that it is not biased toward a big error vs a small error. The Root Mean Squared Error is more strict when it comes to error estimation because the square root sign on the n divides the sum with a lower value, therefore, bigger errors get penalized more than smaller errors. For very small errors, MAE and RMSE will return similar values. Note that $\frac{1}{\sqrt{n}} > \frac{1}{n}$ for $n > 1$. Despite RMSE's higher penalization, it is still used as the default metric [28].

2.4 The activation functions

There are multiple choices for activation functions each with their own properties and domain of application, these are; the sigmoid function, the hyperbolic tangent function, and, but not limited to, the rectified linear unit.

The sigmoid function is the standard introductory activation function. It maps the real number line to numbers between 0 and 1 [**pp.394**, 13].

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad . \tag{18}$$

This is not so useful for data normalization where keeping the signs in order is desired.
Tanh(x) is a modified sigmoid function which instead of mapping between 0 and 1, it maps from -1 to 1.

$$(\frac{1}{1+e^{\frac{-z}{2}}} - \frac{1}{2}) \cdot 2 = 2\sigma(2z) - 1 = \frac{1-e^{z/2}}{1+e^{z/2}} = tanh(z). \tag{19}$$

The motivation behind the use of tanh function is because one might want to map positive numbers between 0 and 1. Tanh also has the advantage of keeping the signs in order. Because tanh keeps the sign in order, this activation function can be used for normalizing data where the output values can in theory be unlimited. The standard method to normalize image values is to divide by 255, which is the maximum value of a pixel. For instance, in stock pricing or traffic volume predictions, the theoretical value can be unlimited or grow very large. For this reason it is unreasonable to normalize the input data by its maximum value or a given number such as 255.

Rectified linear unit (ReLU) and Leaky ReLU are other activation functions which are frequently used. ReLU maps positive numbers linearly, and anything below zero is mapped to zero.

$$f(z) = \begin{cases} z, & z \geq 0, \\ 0, & z < 0. \end{cases} \tag{20}$$

Because ReLU maps any negative values to zero, it has the tendency of killing off neurons, which means these neurons will map anything it touches to zero, making these neurons unusable. Leaky ReLU prevents this by adding some learning to negative numbers as well. This negative mapping is also linear, but the linear slope of the negative mapping is smaller than the slope of the positive mapping.

$$f(z) = \begin{cases} z, & z \geq 0, \\ \alpha z, & z < 0. \end{cases} \tag{21}$$

The motivation behind the use of ReLU is that the sigmoid function and the tanh functions are so called "squishification" functions. This means that the

derivatives of these activation functions quickly tend to zero, because the weight update for both W and b depends on these derivatives, the update may stop, preventing the network from learning. This is known as the vanishing gradient problem. ReLU combat this by mapping linearly instead of squashing it.

ELU is another variant of the ReLU activation function, which works in a similar manner to that of the Leaky ReLU activation function, however, instead of mapping linearly when $z < 0$, it maps exponentially.

$$f(z) = \begin{cases} z, & z \geq 0, \\ \alpha(e^z - 1), & z < 0. \end{cases} \tag{22}$$

Where $\alpha > 0$, usually 0.1 or smaller.

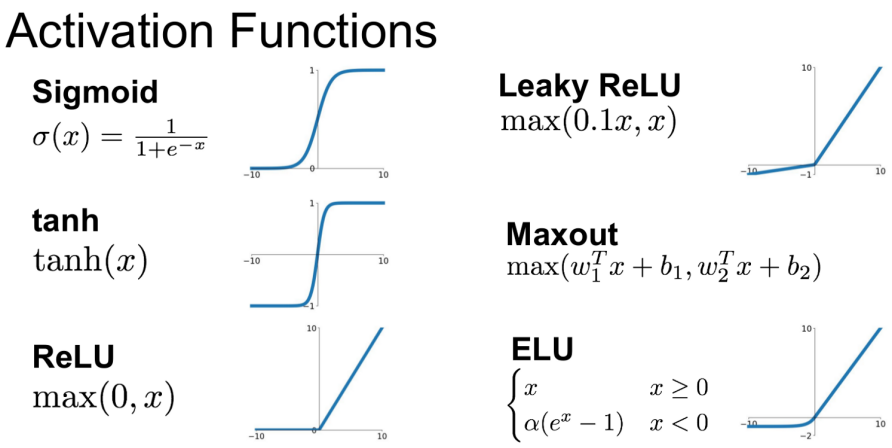


Figure 5: The different types of activation functions. The blue curve shows the mapping from the X-axis to the Y-axis. In our case, z = $Wx + b$ will be the pre-image (X-axis). Image from: https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c.

The derivatives for the activation functions.
Derivative for the standard sigmoid function:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{23}$$

The derivative for the tanh:

$$tanh'(z) = 1 - tanh^2(z). \tag{24}$$

ReLU:

$$f'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0. \end{cases} \tag{25}$$

Leaky ReLU:

$$f'(z) = \begin{cases} 1, & z > 0, \\ \alpha, & z \leq 0. \end{cases} \tag{26}$$

The Exponential Linear Unit:

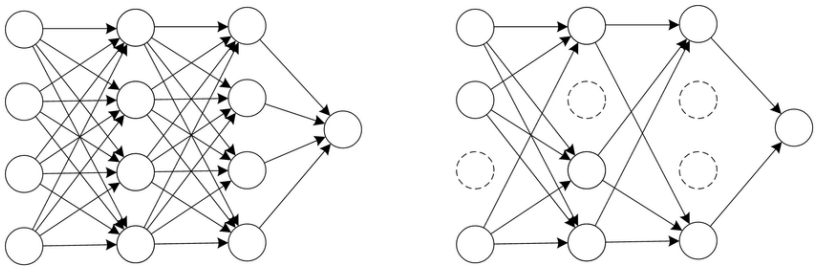
$$f'(z) = \begin{cases} 1, & z > 0, \\ \alpha e^z, & z \leq 0. \end{cases} \tag{27}$$

2.5 Overfitting:

Overfitting is a common phenomenon which causes the network to perform well with the training data, but poorly or even useless with validation data (data which was not used for the training session). According to [14], smaller neural networks contains fewer local minima, but it is easier to converge to these minima. This makes the results from smaller networks to be very volatile as one can be lucky with good convergence. For bigger networks, overfitting is more of a danger. To combat this, several methods are implemented such as dropout, regularization and data-augmentation.

2.5.1 Dropout

Dropout is to simply inactivate certain neurons during the training session to force other neurons to approximate the function better [5]. This prevents co-dependency between the neurons.



(a) Standard Neural Network

(b) Network after Dropout

Figure 6: Dropout randomly chooses neurons with all of its connected parameters to inactivate during a mini-batch training. After each iteration in a mini-batch training, new nodes are randomly selected to be inactivated. The probability p is a hyperparameter which is defined before the training. All neurons have the same probability of dropping out [3].

Each dropout instance (feedforward and backward pass) simulate a new subnetwork which is being trained on the data-set [2]. During the so called training time (feedforward-backward pass) a neuron is active with probability p. This sub-network is then tested and the weights that were not dropped out during the feedforward-backward pass is scaled with a number p (the same value as the probability). The motivation for this is that for a neural network with n parameters, a total of 2^n possible sub-network combinations are possible, but it is unreasonable to try averaging all the sub-network's values after each forward-backward pass considering neural networks have number of parameters in the range of millions. The most effective propability of dropout is generally 0.5, however in this project probability of dropout between 0.2 and 0.25 was sufficient.

2.5.2 Data-Augmentation

Data-Augmentation is the method of applying artificial features to the data-set such as noise which means to add normally distributed random values to the training data. In the real world, data is rarely clean, and there is always imperfections and other impurities. Examples are when training a network to recognize faces. The images used for training can be clearer and have better resolution than real life situations, and it is easy to detect features with the naked eye. In the real world this is not always the case, so in order for the network to learn new features which can only be found in the real world, adding noise to the training data can actually improve the model.

Figure 7: The strength of the noise can be adjusted as a hyperparameter before the training. If an AI is used to detect faces or other objects when it is raining or snowing, it can be difficult to identify these objects correctly. It is therefore important to train the AI so that it can perform its job despite harsher weather conditions, adding noise will help in dealing with this problem. [25]

Figure 8: Example of a real life situation where it can be difficult for the network to detect certain objects. Not only is the smog making it difficult to identify these people, but because of the smog, some of them are covering their faces with masks to keep the smog away. Image from https://www.citymetric.com/horizons/chinas-most-polluted-province-isnt-beijing-1760

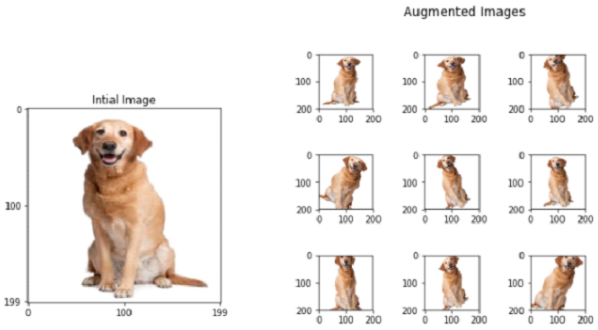


Figure 9: Another type of augmentation used for convolutions is zoom, rotation and distortion and sometimes even changing of colors. Data-augmentation can drastically increase training-data to increase accuracy. One of the rare cases of a "free lunch". These new images let the network detect new features. Not all types of data-augmentation is useful when training a model. A car which is upside down is not very helpful to a model which purpose is to identify cars that which rarely sees upside down cars, however having an augmented image of a car which is tilted a few degrees will be more useful because this can simulate a car travelling uphill or downhill. Flipped images can be good for when trying to detect objects where their rotation is unimportant, take for example a ball. When training a model to determine a crash or a crash prediction, then training on images which are upside down can also be helpful. [27]

2.6 Regularization:

Regularization means that the network model will become more generalized and less complex. More specifically, making the weights more normally distributed, penalizing bigger weights.

In this project the L_2[22] regularization is used:

$$||w||_2^2 = w_1^2 + w_2^2 + ... \tag{28}$$

The following regularization will be added at the end of the weight update from equation (49),

$$W^L - = \Delta W^L \alpha = \delta^L a^{L-1} \alpha + \lambda ||w||_2^2 \quad . \tag{29}$$

Developers at Google [23] mentions the strength of the regularization can be adjusted. Increasing its strength increases the regularization. Making the model more generalized, however, will cause underfitness. Weaken the regularization and the model becomes more complex and will be prone to overfit. The λ is the variance of the weight's values. These weights will be normally distributed for stronger λ values. Setting λ to be lower will reduce the regularization and the weight's values will be more uniform.

In order to find a decent value it is necessary to adjust λ accordingly, [14], [23].

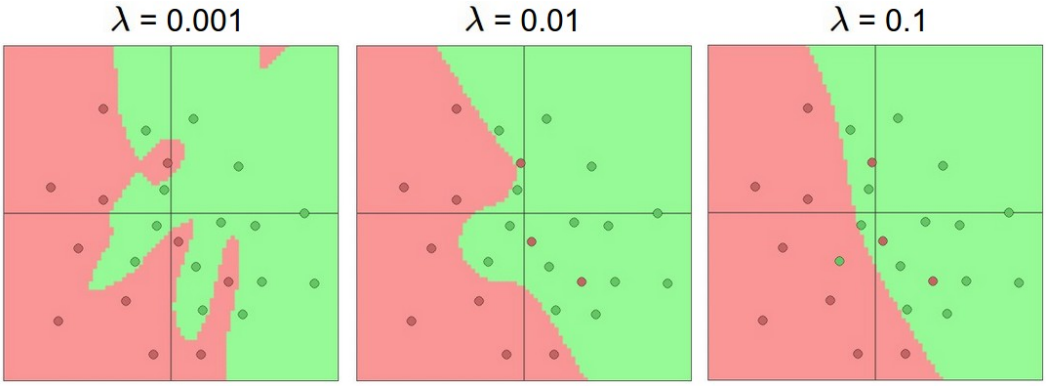


Figure 10: λ is the regularization strength. High regularization can cause underfitting (right image), while too low regularization can cause overfitting (left image). In this case, λ between 0.01 and 0.1 seems to be the better option. The red part of the images is the network's prediction for the red dots, similar for the green part. [14]

2.7 Optimizers

In order to optimize the network for lower error rate and faster convergence, optimizers can be used. These optimizers can change the learning rate during the training and find better (lower) minima.

For this project, the Adam [16] optimizer was used. The motivation for this algorithm is that it uses a combination of two other algorithms such as rmsprop (root mean squared propagation) and gradient descent with momentum. The Adam optimizer is applied to the weight after the gradient has been calculated.

For this optimizer to work certain values must be initialized,

$$V_{dw} = 0, \quad S_{dw} = 0, \quad V_{db} = 0, \quad S_{db} = 0 \tag{30}$$

Recall equation (14),

$$\delta W^l = \frac{\partial C}{\partial W^l}, \quad \delta b^l = \frac{\partial C}{\partial b^l}, \tag{31}$$

where l represents the layer number. The delta functions are the regular update for W^l respective b^l at training step t during the mini-batch training using only stochastic gradient descent.

The mini-batches are smaller subsets of the training data which is randomly scrambled on each new epoch during the training. For each such mini-batch-iteration t, the values V_{dw} etc will be updated. W and b^l will be updated at the end of these iterations.

Momentum update:

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1)\delta W^l, \quad V_{db} = \beta_1 V_{db} + (1 - \beta_1)\delta b^l. \tag{32}$$

RMSprop update:

$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2)(\delta W^l)^2, \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2)(\delta b^l)^2. \tag{33}$$

Note that δW^l and δb^l are element-wise squared and that δW^l is a matrix and δb^l is a vector.

Correction part for the weight and the bias:

$$V_{dw}^c = \frac{V_{dw}}{1 - \beta_1^t}, \quad V_{db}^c = \frac{V_{db}}{1 - \beta_1^t}, \quad S_{dw}^c = \frac{S_{dw}}{1 - \beta_2^t}, \quad S_{db}^c = \frac{S_{db}}{1 - \beta_2^t}. \tag{34}$$

Updating the actual weights:

$$w- = \alpha \frac{V_{dw}^c}{\sqrt{S_{dw}^c} + \epsilon}, \quad b- = \beta \frac{V_{db}^c}{\sqrt{S_{db}^c} + \epsilon}. \tag{35}$$

Recommended values for hyperparameters:

$$\alpha = 0.001, \quad \beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 10^{-8}. \tag{36}$$

α needs to be tuned and β_1 and β_2 can also be tuned, but often this is not needed.

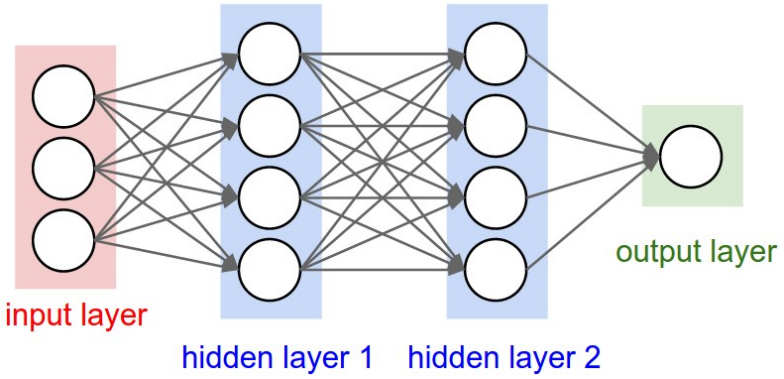


Figure 11: A fully connected neural network without dropout. The hidden layers 1 and 2 are known as dense layers. This is the most basic neural network and is used as an introductory step when learning about neural networks, despite being an introductory step it is widely used in combinations of other type of layers such as LSTM and CNN, usually at the end of a neural network. The arrows can be seen as the weights, therefore the weights are all matrices whereas the bias is a vector. Image from: http://cs231n.github.io/convolutional-networks/.

2.8 Fully connected:

2.8.1 Feedforward

To understand the derivatives in equation (14) and how W and b can be adjusted, it is important to know which activation function to use and how many hidden layers are used.

Figure 11 shows a classic fully connected neural network. In the picture there is one input layer which can be in multiple dimensions. Two hidden layers where the calculations are made and one output layer, this output layer is then compared to a desired output (y or Y), this output is usually defined as a^L whereas the input is defined as x or X.

The fully connected neural network is a multidimensional regression. Using no hidden layers and you will have a linear regression: $y_i = mx_i + b + \epsilon_i$ where m is an unknown parameter which is the slope of the linear function, x_i is the given input also known as the independent variable, y_i is the dependant variable which is also known, b is the intercept which is unknown. And finally ϵ_i is the error which is normally distributed $\mathcal{N}(\mu, \sigma^2)$ [12]. Using one hidden layer and it is possible to approximate any function in $\mathbb{R}$ [8].

Using an activation function the hidden layer is defined as

$$a = \sigma(Wx + b) \tag{37}$$

where a is the output and σ can be any activation function. In other words, a are values for the first hidden layer, W is the weight working on x (input) and b is known as the bias.

For the second hidden layer:

$$a^{L-1} = \sigma(W^{L-1}a + b^{L-1}) = \sigma(W^{L-1}\sigma(Wx + b) + b^L). \tag{38}$$

And the output layer:

$$a^L = \sigma(W^L a^{L-1} + b^L) = \sigma(W^L\sigma(W^{L-1}a + b^{L-1}) + b^L). \tag{39}$$

The L in the superscript represent the final layer.

Because the weights W are initiated with random numbers (generally between -1 and 1), the output will be far from correct to the desired output y. The error between the output a^L and y is defined as $(a^L - y)^2$ and the loss is defined as $\frac{1}{n}\sum_i^n (a_i^L - y_i)^2$ for this particular training example. This is the mean squared error for one training example. n is the number of elements in the vector/ matrix a^L and y. Note here that a^L and y are of same dimensions and both can be a vector or a matrix.

The goal of neural networks is to adjust the parameters W and b (also known as the weights and biases) in order to get a satisfactory output a^L. By minimizing the error for all training instances (see equation (13)), the training is complete. In practical terms, the total loss will virtually never reach zero, it will only be 'close enough' to zero.

It is easy to adjust the weights so that the network will yield good result for only one training example, it is therefore necessary to adjust the weights with respect to multiple training examples. Otherwise the weights would only be adjusted for the last training-data used. The training-data is then split up in batches, these batches have equal number of training-data in them. Training on these batches is one epoch. After each epoch, the training-data is scrambled randomly and the training begins again.

2.8.2 Backpropagation

In order to adjust the weights and biases correctly of equations (38) and (39) it is necessary to know how much to adjust them. In general it is possible to adjust these manually, but it would be unreasonable to do so with a bigger neural network such as the network in Figure (11). However, for a small neural network which only has a few nodes (neurons) in it and one hidden layer this would not be too difficult to do. (Imagine adjusting the equalizer on your sound system to get a satisfactory audio).

To minimize the total loss function (see equation (13)) by adjusting the weights and biases from equation (39), the cost function (metric) must first be defined. In this paper the mean squared error will be the default metric to be used. The cost function is defined in equation (11) and the metric is defined in equation (12). From the feedforward part, there are multiple functions within one and another, the chain rule is used to find the derivatives from equation (14). These derivatives tells us in which direction to adjust the parameters. This method of optimizing the network is known as gradient descent. Let $f(g(x))$ and $g(x)$ be two functions that which depends on x, then the definition of the chain rule states:

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)}\frac{dg(x)}{dx}. \tag{40}$$

From the chain rule, it is then necessary to start with the derivatives at the very end and then work your way to the beginning of the network's equations. This makes it possible to adjust the parameters (weights and biases) which are located at the end. In other words, the final parameters used from the feedforward is adjusted first during the backpropagation process.
Recall that the functions for the final layer are,

$$z^L = W^L a^{L-1} + b^L, \tag{41}$$

$$a^L = \sigma(z^L), \tag{42}$$

$$C = (a^L - y)^2. \tag{43}$$

Then applying the chain rule for equation (14) to get the gradient of the weights,

$$\frac{\partial C}{\partial W^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial W^L}, \tag{44}$$

and the gradient for the bias,

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial b^L}. \tag{45}$$

So,

$$\frac{\partial C}{\partial a^L} = 2(a^L - y), \quad \frac{\partial a^L}{\partial z^L} = \sigma'(z^L), \quad \frac{\partial z^L}{\partial W^L} = a^{L-1}, \quad \frac{\partial z^L}{\partial b^L} = 1. \tag{46}$$

Because σ can be any activation function, σ' will be its derivative.

Then define the delta function δ which is useful for computing the next chain of derivatives,

$$\delta^L = \frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} = 2(a^L - y) \odot \sigma'(z^L). \tag{47}$$

Note:

$$\delta^L = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} = 2(a^L - y) \odot \sigma'(z^L) = \nabla_{a^L} \odot \sigma'(z^L). \tag{48}$$

This product is known as the delta function and $\odot$ is defined as element-wise multiplication.

Putting all this together:

$$W^L - = \Delta W^L \alpha = \delta^L a^{L-1}\alpha, \quad b^L - = \Delta b^L \alpha = \delta^L \alpha, \tag{49}$$

where ΔW^L is an outer product of two vectors(δ^L, a^{L-1}) where this is a rank 1 update, similar to that of the Broyden's rank 1 update which instead of updating the weight with a rank 1 matrix, it updates the Jacobian with a rank 1 matrix, both methods are used for finding the minima in their respective optimization problems. ΔW^L can also be a dot product matrix from two other matrices instead of an outer product, this is the case when all a^l (including the input X) and y are matrices.
Δb^L is just a vector, but if the input and output x and y are matrices, then b^L and Δb^L must also be a matrix. The minus sign is used because we want the slope of the negative descent, and finally α is the learning rate. This decides how big of a step the weights will take. Too big and the weights will miss the minima value, too small and it will take a very long time to train the network. The most common learning rates are 10e-4 or 10e-5.

The next step is to adjust the weights for the next layer (L-1) also known as the first hidden layer (with respect to the backpropagation process). Simply use the same principle of the chain rule as before.

To change the weight matrix W^{L-1} just repeat the previous chain rule steps:

$$\frac{\partial C}{\partial W^{L-1}} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial W^{L-1}}. \tag{50}$$

Recall that,

$$a^{L-1} = \sigma(z^{L-1} = W^{L-1}a^{L-2} + b^{L-1}), \tag{51}$$

and

$$a^L = \sigma(z^L = W^L a^{L-1} + b^L). \tag{52}$$

This gives,

$$\frac{\partial z^L}{\partial a^{L-1}} = W^L, \quad \frac{\partial a^{L-1}}{\partial z^{L-1}} = \sigma'(z^{L-1}), \quad \frac{\partial z^{L-1}}{\partial W^{L-1}} = a^{L-2}. \tag{53}$$

Because the dimensions of the vectors and matrices may not coincide between $\delta^L = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}$ and $\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial W^{L-1}}$ it is possible to take a transpose and changing the placement of the weights as the following,

$$\delta^{L-1} := \left((W^L)^T\delta^L\right) \odot \sigma'(z^{L-1}), \tag{54}$$

where δ^L is defined as previously.

And for a general formula,

$$\delta^l := \left((W^{l+1})^T\delta^{l+1}\right) \odot \sigma'(z^l). \tag{55}$$

After producing the relevant δ function,

$$\Delta W^{L-1} = \delta^{L-1}(a^{L-2})^T. \tag{56}$$

If only one hidden layer is used, then $a^{L-2} = X$ is the input.
And finally updating the weight and bias,

$$W^{L-1} -= \Delta W^{L-1}\alpha, \quad b^{L-1} -= \delta^{L-1}\alpha \tag{57}$$

Then continue in this fashion for whatever hidden layers that may be next.

Let's see this with an example: Say we have a neural network with one hidden layer. The dimensions for each layer is: 5×1 for the input, 7×1 for the hidden layer and finally 3×1 for the final output layer.
Now to find the dimensions of the weights:

$$a = \sigma(WX + b). \tag{58}$$

Where a have dimensions 7×1. X is the input which has dimensions 5×1 this implies that the dimensions for W is 7×5 and the bias b have dimensions 7×1 as well.

The final output layer:

$$a^L = \sigma(W^L a + b^L), \tag{59}$$

has dimensions 3×1 which means W^L have dimensions 3×7 and the bias b^L has dimensions 3×1.

Using equation (45) the dimensions will be: 3×1, 3×1, 7×1. These dimensions do not coincide, in order to solve this, we must first transpose $\frac{\partial z^L}{\partial W^L} = a$. Then ΔW^L have been defined and have dimensions 3×7.

To update the next weight $W = W^{L-1}$, use equation (50). This gives dimensions: 3×1, 3×1, 3×7, 7×1, 5×1.

To solve this problem simply use equation (54) and then update ΔW^{L-1} accordingly.

Equation (54) in dimension form:

$$\delta^{L-1} = (3 \times 7)^T 3 \times 1 \odot 7 \times 1 = 7 \times 1. \tag{60}$$

Then δ^{L-1} is a matrix multiplied with $(a^{L-2})^T = X^T = (\frac{\partial z^{L-1}}{\partial W^{L-1}})^T$ to get dimension $(7 \times 1) \times (5 \times 1)^T = 7 \times 5$ which are the dimensions for W and we're done. Note that in this example, a^{L-2} is the input X

$$\frac{\partial C}{\partial W} = \delta^{L-1}(\frac{\partial z^{L-1}}{\partial W})^T = \delta^{L-1} X^T. \tag{61}$$

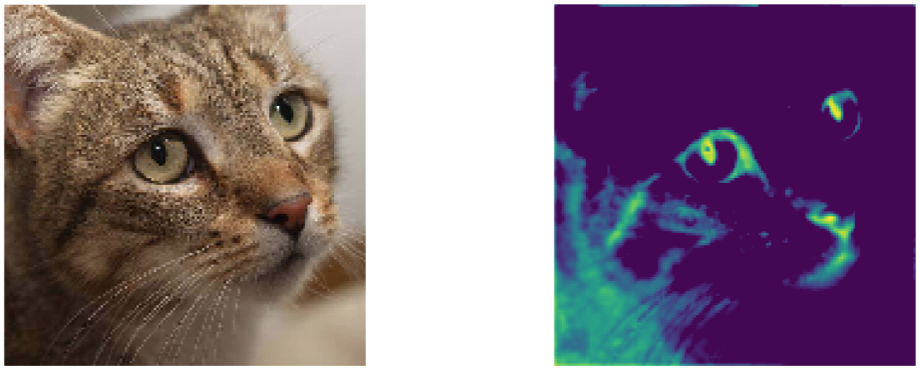
2.9 Convolutional Neural Network

The motivation behind using convolution layers is to reduce the numbers of parameters given from the original input image. The convolutional process will also summarize an image, these summarizations are used for classifications, different filters will summarize differently on the same image. Another property of convolutional layers is that it is possible to sharpen, blur and even detect borders of objects in the images. An input image of size 250×250 can cause the network to have millions even billions of parameters which all need to be adjusted, this is very cost intensive for the computer and takes a long time to train. State of the art neural networks may take up to weeks to train.

2.9.1 Feedforward

For convolutional layers, the feedforward-process works in a similar manner to that of the fully connected layers. A weight is multiplied with an input image with a bias and this function is then activated using for example the ReLU function. However the weight matrix is not multiplied with the input using the dot product, but rather elementwise multiplication. Also, the weight matrix is not the same dimensions as the input image, rather, the dimensions of the weight matrix coincide with a submatrix of the input image. This weight matrix is known as a filter [10].

A filter is a $n \times n$ matrix which have weights in its entries. This filter is element-wise multiplied with a sub-matrix (of same size as the filter) of the original input matrix. Imagine a square cover hovering above the picture. This filter will go from left to right until the filter's right edge hits the input image's right edge. Then the filter will reset to its original position, however, this time it will take a step downwards and start the process all over again.



(a) Original image. (b) Convoluted image.

Figure 12: A convolution of the original input image. Notice how the nose and eyes are highlighted. This is what the filters do, it looks for certain features of the original input image and highlights them while dimming other features [10].

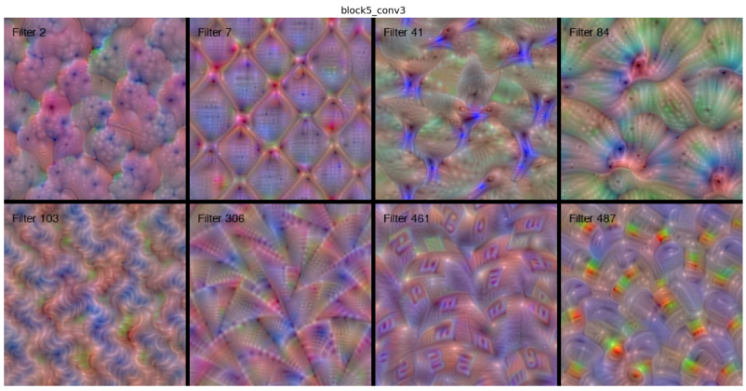


Figure 13: A collection of filters after a training session. The filters become increasingly abstract for each convolution. The dimensions of each filter is usually square, and their sizes can vary from anywhere between 2 to 100 or even bigger. [10]

Pooling happens after the convolution part, this process reduces the dimensionality of the convoluted image. There exists different pooling methods, one of which is the maxpooling, a very common pooling technique similar to the convolution process.

Padding is when a section of the filter starts outside the input image [6] p.12. A padding of one means that one row and one column of the filter will be outside the input image. The same principle holds for when the filter scans the image. Instead of stopping the process when the two edges coincide, the filter will continue until the rightmost column of the filter is outside the input image.

The stride determines how far this filter jumps. A stride of one is usually the default value, this means that the filter will only do one step at a time.

The filter is a set of weights that is element-wise multiplied by the input image which is covered by the filter. For a 2×2 filter there are 4 weights.

Example: Filter 1 is a 3×3 matrix and our input image is a 5×5 matrix.

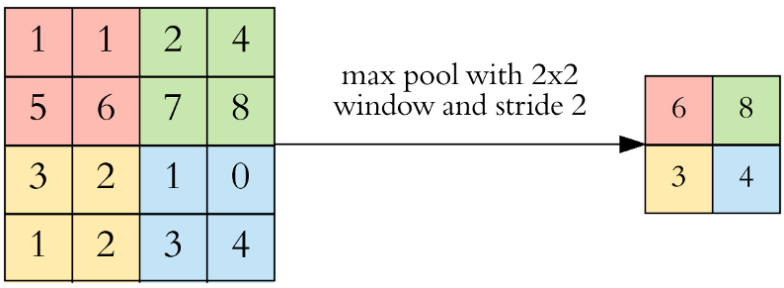


Figure 14: The maxpool takes the biggest value in each 2x2 window, [10].

We wish to convolute this image to a 3×3 matrix.

Filter 1 has weights W_1 to W_9. The first pixel (the topleft) is connected to W_1. The next pixel is connected to W_2, the pixel just below pixel 1 is pixel 6 which is connected to W_4 and pixel 7 which is just to the right of pixel 6 is connected to W_5.

By multiplying element-wise the weights with the pixels and then summing this product, z_1 is created. z_1 is the first pixel (top-left) of our convoluted image z also known as the feature map (which will become a 3x3 matrix).

Note now that W_1 is only used on the 3×3 sub-matrix with respect to our 5×5 input image. This sub-matrix begins in the top-left corner of the 5×5 input image. Now similar for W_2 which is only affects the pixels on the top-middle of the input image. W_3 only affects the pixels on the top-right of our 5×5 input image. And similar for the other weights in this filter. And finally W_9 affects the pixels which is the 3x3 sub-matrix on the bottom-right of our input image,

$$z_1 = W_1 p_1 + W_2 p_2 + W_3 p_3 + W_4 p_6 + W_5 p_7 + W_6 p_8 + W_7 p_{11} + W_8 p_{12} + W_9 p_{13}. \quad (62)$$

The resulting matrix from the convolution will have dimensions O_d and is called a feature map:

$$O_d = \frac{n_d + 2p - k_d}{s} - 1 \quad . \quad (63)$$

Where O_d is the output dimension for d representing the rows or the columns or both, n_d is the dimension for the input image, p is the padding, k_d is the filter size and s is the stride.

To keep the dimensions of the input image and the feature map the same, the following formula is used:

$$p = \frac{s(O+1) - O + k}{2} \quad . \quad (64)$$

Note that k, which is the filter size, must be consistent when doing the feedforward as well as full convolution for the backpropagation.

In our example we had the constants defined: $n = 5$, $p = 0$, $k = 3$, $s = 1$. This gives us the output size to be 3.

1	1	1	0	0
0	1	1	1	0
0x1	0x0	1x1	1	1
0x0	0x1	1x0	1	0
0x1	1x0	1x1	0	0

4	3	4
2	4	3
2		

Figure 15: In the green area: Image values to the left and filter weights to the right. The red image is the feature map. When no padding is present, the dimensions of the feature map is smaller than the original input. This depends on the stride and the filter size. [7]

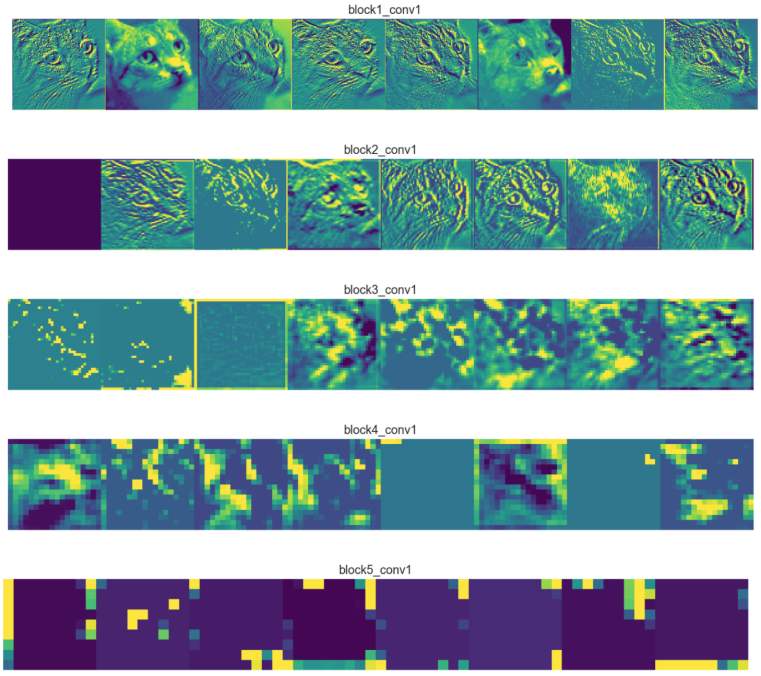


Figure 16: The feature maps after a training session. This map will highlight certain features of the input image. These are the convoluted images after one convolution. Notice how the size of the dimensions is reduced for each convolution (rows). [10].

2.9.2 Backpropagation for convolutional layers

The same principal of propagating backwards is used for convolutional layers just as in fully connected layers.

Assuming we have the definition of the delta function as before:

$$\delta^L \frac{\partial z}{\partial W} := \frac{\partial C}{\partial W} = 2(a^L - y) \odot \sigma'(z) \odot \frac{\partial z}{\partial W} \tag{65}$$

The difference between convolutional and fully connected backpropagation is that no transpose is used nor is the dot product used. Only elementwise multiplication is used for backpropagation in the convolutional layers, or equivalently a so called full-convolution is used, more on this later.

δ^L is defined as before, but now it is needed to derive $\frac{\partial z}{\partial W}$

Recall that $z = \sum_i a_i^{L-1} W_i$. By taking the partial derivative of z wrt W_i what is returned is the input image, but only the pixels that which were affected by W_i for some i. This will return a sub-matrix consisting of the input image pixels which were affected by W_i.

From the example above, the sub-matrix would look like this:

$$X_1^L = \frac{\partial z^L}{\partial W_1^L} = \begin{bmatrix} a_1^{L-1} & a_2^{L-1} & a_3^{L-1} \\ a_6^{L-1} & a_7^{L-1} & a_8^{L-1} \\ a_{11}^{L-1} & a_{12}^{L-1} & a_{13}^{L-1} \end{bmatrix}. \tag{66}$$

This is the matrix of pixels which were affected by W_1. Similar matrices will be made for the other W_2, W_3....

To change the weight W_i (which is only a real number) it is required to sum $\frac{\partial C}{\partial W_i^L}$ element-wise.

One can flatten δ^L and $X_{j,i}$ first.

$$\frac{\partial C}{\partial W_i^L} = \Delta W_i = \frac{1}{9} \sum_{j=1}^{9} (\delta_j^L \odot X_{j,i}) \tag{67}$$

With $X_{j,i}$ denoting the submatrix i for the weight W_i and j is the element position. Then update the weight: $W_i - = \Delta W_i \alpha$ where α is the learning rate. Usually at 10e-4 or less.

This was an update for the first layer. Because the dimensions of the first delta: δ^L will not be the same dimensions as $\frac{\partial z^L}{\partial a^{L-1}}$ from equation (50), it is required to do a full convolution between the filter weights W^L and δ^L, [1]. Full convolution means to do convolution as in the feedforward phase with padding. The padding constant p is the difference in dimensions between the input image and the feature map. Because our feature map is of dimension 2×2 and the input image is of dimensions 3×3, the padding will be defined as $p = 1$.

This will return the partial derivatives with proper dimensions:

$$\frac{\partial C}{\partial a^{L-1}} = \frac{\partial C}{\partial a^L} \frac{\partial a^L}{\partial z^L} \frac{\partial z^L}{\partial a^{L-1}} = \delta^L \frac{\partial z^L}{\partial a^{L-1}} \quad . \tag{68}$$

For simplicity let the feature map z^L be a 2×2 matrix and our input image a^{L-1} a 3×3 matrix. The full convolution: $\frac{\partial C}{\partial z^L}$ with $\frac{\partial z^L}{\partial a^{L-1}}$ Where both are 2×2 matrices (in this example) will return a 3×3 matrix. But first it is necessary to define $\frac{\partial z^L}{\partial a^{L-1}}$

From the feedforward part z^L is defined as such:

$$z^L = \begin{bmatrix} W_1 a_1^{L-1} + W_2 a_2^{L-1} + W_3 a_4^{L-1} + W_4 a_5^{L-1} & W_1 a_2^{L-1} + W_2 a_3^{L-1} + W_3 a_5^{L-1} + W_4 a_6^{L-1} \\ W_1 a_4^{L-1} + W_2 a_5^{L-1} + W_3 a_7^{L-1} + W_4 a_8^{L-1} & W_1 a_5^{L-1} + W_2 a_6^{L-1} + W_3 a_8^{L-1} + W_4 a_9^{L-1} \end{bmatrix}. \tag{69}$$

Taking the derivative of z^L with respect to a_1^{L-1}:

$$\frac{\partial z^L}{\partial a_1^{L-1}} = \begin{bmatrix} W_1 & 0 \\ 0 & 0 \end{bmatrix}. \tag{70}$$

Taking the derivative of z^L with respect to a_2^{L-1}:

$$\frac{\partial z^L}{\partial a_2^{L-1}} = \begin{bmatrix} W_2 & W_1 \\ 0 & 0 \end{bmatrix}. \tag{71}$$

Continue like this for all a_i^{L-1}.

Recall that:

$$\frac{\partial C}{\partial z^L} = \delta^L = \begin{bmatrix} \frac{\partial C}{\partial z_1^L} & \frac{\partial C}{\partial z_2^L} \\ \\ \frac{\partial C}{\partial z_2^L} & \frac{\partial C}{\partial z_4^L} \end{bmatrix}. \tag{72}$$

Then:

$$\frac{\partial C}{\partial a_1^{L-1}} = \begin{bmatrix} \frac{\partial C}{\partial z_1^L} & \frac{\partial C}{\partial z_2^L} \\ \\ \frac{\partial C}{\partial z_2^L} & \frac{\partial C}{\partial z_4^L} \end{bmatrix} \odot \frac{\partial z^L}{\partial a_1^{L-1}} = \begin{bmatrix} \frac{\partial C}{\partial z_1^L} W_1 & 0 \\ \\ 0 & 0 \end{bmatrix},$$

and sum up each of these 2×2 matrices which will be an entry in the bigger 3×3 matrix. The summing of the entries in this 2×2 matrix is the same as the summing in the feedforward convolution process.

The derivative $\frac{\partial C}{\partial a^{L-1}}$ will look like this:

$$\frac{\partial C}{\partial a^{L-1}} = \begin{bmatrix} \frac{\partial C}{\partial z_1^L} W_1 & \frac{\partial C}{\partial z_2^L} W_1 + \frac{\partial C}{\partial z_1^L} W_2 & \frac{\partial C}{\partial z_2^L} W_2 \\ \frac{\partial C}{\partial z_3^L} W_1 + \frac{\partial C}{\partial z_1^L} W_3 & \frac{\partial C}{\partial z_4^L} W_1 + \frac{\partial C}{\partial z_3^L} W_2 + \frac{\partial C}{\partial z_2^L} W_3 + \frac{\partial C}{\partial z_1^L} W_4 & \frac{\partial C}{\partial z_4^L} W_2 + \frac{\partial C}{\partial z_2^L} W_4 \\ \frac{\partial C}{\partial z_3^L} W_3 & \frac{\partial C}{\partial z_4^L} W_3 + \frac{\partial C}{\partial z_3^L} W_4 & \frac{\partial C}{\partial z_4^L} W_4 \end{bmatrix}. \tag{73}$$

This is equivalent of taking the full convolution between W^r and $\frac{\partial C}{\partial z^L}$ to get $\frac{\partial C}{\partial a^{L-1}}$.

$$W = \begin{bmatrix} W_1 & W_2 \\ W_3 & W_4 \end{bmatrix} \quad . \tag{74}$$

Rotate this matrix 180 degrees before doing the full convolution,

$$W^r = \begin{bmatrix} W_4 & W_3 \\ W_2 & W_1 \end{bmatrix} \quad . \tag{75}$$

Then the full convolution of equation (72) and equation (75) will be the same as equation (73). Recall that this full convolution requires a padding $p = 1$. Note here that it is W^r that works as the filter.
Then using the same principle as before for the next two partials:

$$\frac{\partial C}{\partial W_i^{L-1}} = \frac{\partial C}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial z^{L-1}}{\partial W_i^{L-1}}. \tag{76}$$

Where $\frac{\partial a^{L-1}}{\partial z^{L-1}} = \sigma'(z^{L-1})$ and $\frac{\partial z^{L-1}}{\partial W_i^{L-1}} = X_i^{L-1}$. Thus all dimensions will coincide.

2.10 Recurrent neural network:

Recurrent neural network is a type of network which is used for time-dependant machine learning. For each time-step, an input X_t is given with a corresponding desired output Y_t, where t is the current time-step.

The idea behind this layer is that it can 'remember' data from previous time-steps. There is a neuron inside of this layer for all time-steps. This neuron is like any neurons previously discussed, it has a weight, an input and a bias.

It is called recurrent because this neuron is also an input for the next time-step. As a result,

$$h_t = \sigma(W^L a_t + b_h), \tag{77}$$

is the output at time-step t and W^L is the weight applied on a_t.

The layer A from Figure (17) is defined as:

$$a_t = \sigma(W^{L-1} X_t + W^T h_{t-1} + b^{L-1}) \tag{78}$$

Where L in the superscript denotes the final layer for this time-step and W^T is the same weight for all timesteps which is working on the recurrent neuron h_{t-1}. Note that σ can be any activation function.
A time dependant neural network aka time series neural network uses backpropagation to reduce the error just like the CNN and the fully connected (FC).

The final activation function in a time-series prediction can be of different types. One is the standard Mean Square Error, but another one uses probability, this activation function is called Softmax.

For a time series model using Softmax, it will guess with a probability which action to take or which input to classify for each time step. Thus each output (from the activation function) will be a probability ranging from 0 to 1 and the sum of the Softmax vector will always be 1 for any timesteps. Note: Other outputs than probabilities can also be computed, such as a vector output. This is for predicting actual values (such as number of cars passing an intersection), see Mean Squared Error.

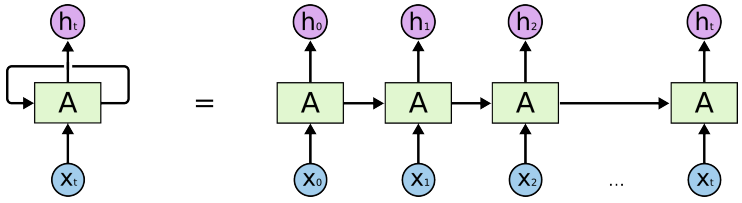


Figure 17: The RNN. Unrolled to the left and rolled to the right. The recurrent weight (the arrows between the A's) is the same for all timesteps. Similar for the weights between x^t and h^t, these weights are also shared for any timestep.

2.11 LSTM

A variant of the recurrent architecture is the Long Short-Term Memory type layer or LSTM for short [26]. This is an architecture which is based off of the memory of a cell. The LSTM is a recurrent neural network which remembers past time series data for a limited time. This type of network is excellent at predicting future data set such as traffic prediction and even stock market predictions. The downside of the LSTM is that it is more resource intensive and it will therefore take a longer time to train these types of network.

The layer is divided into blocks for each timestep much like the recurrent layer shown above. There are three inputs for each block (one input for the initial block). The first input is the current timestep data, every block has this type of input. The other two inputs comes from the previous LSTM block (which is the previous timestep).

Each LSTM block have three outputs. The first output is compared with the real data at this timestep, and the other two outputs will be given to the next LSTM block (except for the final block which only has one output, the output which is compared to the real data at the final timestep), these two outputs which are sent to the next block are both modified in their own way, so they are not the same.

Another interesting and important property of the LSTM is that each LSTM block can hold information for more than one timestep. For instance. A neural network using the LSTM layer with 10 timesteps (10 LSTM blocks) can in practice predict time-series for 20 timesteps. This fact is very useful for training a neural network as the LSTM layers are expensive to optimize. In fact, time series data-sets may have tens of thousands of timesteps which makes it unreasonable to build a network where there are an equal amount of LSTM blocks as there are data timesteps. Therefore shrinking the recurrent neural network by rearranging the data-set is preferred, especially for data-sets with many timesteps. An example of shrinking the neural network by rearranging the data-set can be, for a data-set with 1000 timesteps, and for each time-step there is a corresponding vector of size 10. This gives the data-set's dimension to be (1000, 10). By reshaping this matrix to (100,10,10) the neural network now only requires 100 timesteps, the 2nd entry represents the number of samples for a given timestep and the final entry represents the values for a given sample. Each LSTM block will now be able to hold 10 different timesteps values.

Because each LSTM block has at least one output and input for each timestep (the output which is compared with the real data and the input which takes timestep data), LSTM layers can be stacked on top of each other creating a grid pattern with LSTM layers. Doing this can increase accuracy at the cost of slower training sessions See Figure (29).

2.11.1 Activation and cost functions for LSTM: Softmax and Cross Entropy

Besides using Mean Squared Error which was defined previously, the Softmax activation function [9] can be used for classification when using a RNN type network. Instead of returning a vector or a matrix as real values such as the number of cars for a certain time-step, it returns a probability vector with values between 0 and 1 for each timestep. In short, the softmax activation function takes an unnormalized vector valued function z^L and normalizes it to a probability vector.

$$(a^L)^{(t)} = \frac{e^{(z)^{(t)}}}{\sum_i^N e^{(z_i)^{(t)}}} \tag{79}$$

where t denotes the timestep, i is the entry position and N is the number of entries in the vector $(a^L)^{(t)}$ and $(z^L)^{(t)}$. Therefore, summing over all the $(a_i^L)^{(t)}$'s with respect to i will return 1.

$$\sum_i^N (a_i^L)^{(t)} = 1 \tag{80}$$

Equation (79) gives the change of $(a^L)^{(t)}$ wrt $(z^L)^{(t)}$

$$\frac{\partial (a^L)^{(t)}}{\partial (z^L)^{(t)}} = ((a^L)^{(t)} \odot (1 - a^L)^{(t)}) \tag{81}$$

The error function [24] at timestep t is known as cross entropy

$$C^{(t)} = -\sum_i^N y_i^{(t)} \odot \log((a_i^L)^{(t)}) \tag{82}$$

where $(y_i)^{(t)}$ is the desired output at timestep t.

This gives the change of the cost function wrt a^L

$$\frac{\partial C}{\partial (a^L)^{(t)}} = -(\frac{y_1}{a_1^L}, \quad \dots \quad , \frac{y_N}{a_N^L}) \tag{83}$$

Notice that both y and a^L are in vector form, hence the element wise multiplication symbol. If, however, the desired output $y^{(t)}$ is a hot vector such as $y^{(t)} = (1, 0, 0)$ then equation (82) simply becomes

$$C^{(t)} = -\log((a_1^L)^{(t)}) \tag{84}$$

The total loss for the Softmax is summed for all timesteps

$$C_T = \frac{1}{n} \sum_{t=1}^{n} C^{(t)} \tag{85}$$

if $(a_1^L)^{(t)} = 1$ then the cost function of equation (84) is 0.

The other usual cost function is the mean square error which has been defined previously.
Loss:

$$C_T = \frac{1}{2N} \sum_{x}^{N} \|a^L(x) - y(x)\|^2 \tag{86}$$

Cost:

$$C = (a^L - y)^2 \tag{87}$$

Note that this a^L is not the same as the one in equation (79)

$$a^L = \sigma(W a^{L-1} + b) \tag{88}$$

Where σ in equation (88) is the typical activation functions such as tanh or sigmoid.

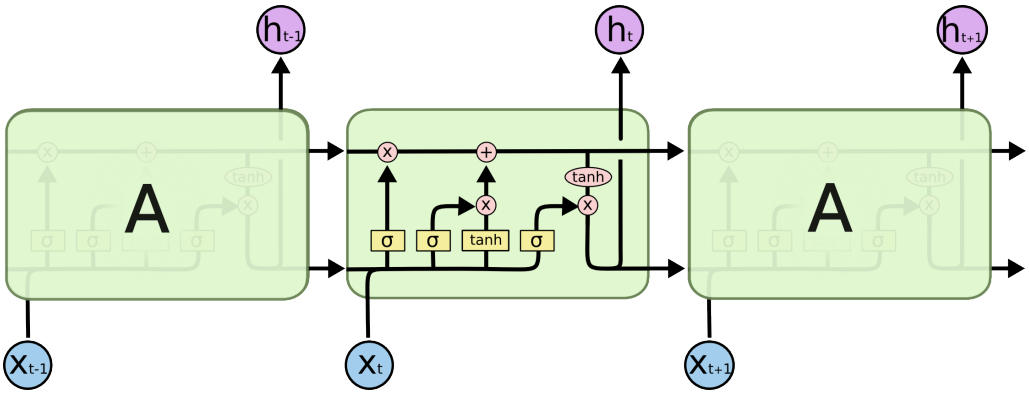


Figure 18: The architecture of the LSTM layer. This unit uses two different types of activation functions which are spread throughout. The first activation functions used is the sigmoid and the second is the tanh. The $\times$ and $+$ signs are element-wise multiplication and addition.

2.11.2 Feedforward of the LSTM:

The first activation function in this picture (bottom-left) is the sigmoid function. This is the "forget gate". Previous data h_{t-1} is added with the input data X_t, a bias is added and then activated. This tells us how much of the previous data will be remembered or forgotten.

The function looks like this:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{89}$$

The next step is to simply add new information to the current LSTM layer's information. The sigmoid part (the 2nd sigmoid function in this layer) decides which values will be updated. The tanh function gives new information to these values.

The sigmoid function:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{90}$$

The tanh function which is the input activation function:

$$a_t = tanh(W_a[h_{t-1}, x_t] + b_a) \tag{91}$$

Then using these new functions it is possible to update this layer's information. This is represented by the horizontal line on top.

$$C_t = f_t \odot C_{t-1} + i_t \odot a_t \tag{92}$$

The final activation functions decides the output h_t and this output will then be given to the next LSTM layer.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{93}$$

$$h_t = o_t \odot tanh(C_t) \tag{94}$$

C_t is defined as the internal state and h_t is defined as the output.

$$gates_t = \begin{pmatrix} f_t \\ i_t \\ a_t \\ o_t \end{pmatrix}, \quad W = \begin{pmatrix} W_f \\ W_i \\ W_a \\ W_o \end{pmatrix}, \quad U = \begin{pmatrix} U_f \\ U_i \\ U_a \\ U_o \end{pmatrix}, \quad b = \begin{pmatrix} b_f \\ b_i \\ b_a \\ b_o \end{pmatrix},$$

(95)

For clarity, note that $W_i[h_{t-1}, x_i]$ is a truncated version of $W_i x_t + U_i h_{t-1}$. Where W and U are weights for the input respective output from previous time-frame.

There exists variants of the LSTM style network, but this is the standard architecture.

$$g_t = \begin{pmatrix} f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ a_t = \sigma(W_a x_t + U_a h_{t-1} + b_a) \\ o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \end{pmatrix}. \tag{96}$$

2.11.3 Backpropagation for LSTM:

The cost function will now be defined as E instead of C for the LSTM part. Just like any other network using backpropagation, gradient descent will be used to calculate the derivatives for changing the weights. The chain of derivatives will now be based on time-frames instead of simply derivative chains as before.

An example of this:
Before:

$$\frac{\partial C}{\partial W^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial W^L} \tag{97}$$

After:

$$\frac{\partial E_t}{\partial W_o} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial z_{o,t}}\frac{\partial z_{o,t}}{\partial W_o} = \\ 2(h_t - y_t) \odot tanh(C_t) \odot \sigma'(z_{o,t}) \odot x_t \tag{98}$$

Recall that $h_t = o_t \odot tanh(C_t)$. $\sigma'(z)$ is the derivative of the activation function. And x_t is the input for this timestep.
Because there is a 2nd weight which is affecting the previous output namely h_{t-1} a second chain of derivative is needed. Simply replacing W with U:

$$\frac{\partial E_t}{\partial U_o} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial z_{o,t}}\frac{\partial z_{o,t}}{\partial U_o} = \\ 2(h_t - y_t) \odot tanh(C_t) \odot \sigma'(z_{o,t}) \odot h_{t-1} \tag{99}$$

Note that these were the weights for $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$.
Where $z_{o,t} = W_o[h_{t-1}, x_t] + b_o = W_o x_t + U_o h_{t-1} + b_o$

Gradient descent works similar for the other activation functions:

$$\frac{\partial E_t}{\partial W_f} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial z_{f,t}}\frac{\partial z_{f,t}}{\partial W_f} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot C_{t-1} \odot \sigma'(z_{f,t}) \odot x_t \tag{100}$$

Now with respect to U_f:

$$\frac{\partial E_t}{\partial U_f} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial z_{f,t}}\frac{\partial z_{f,t}}{\partial U_f} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot C_{t-1} \odot \sigma'(z_{f,t}) \odot h_{t-1} \tag{101}$$

Gradient descent for W_i

$$\frac{\partial E_t}{\partial W_i} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial z_{i,t}}\frac{\partial z_{f,t}}{\partial W_i} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot a_t \odot \sigma'(z_{i,t}) \odot x_t \tag{102}$$

With respect to U_i

$$\frac{\partial E_t}{\partial U_i} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial z_{i,t}}\frac{\partial z_{f,t}}{\partial W_i} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot a_t \odot \sigma'(z_{i,t}) \odot h_{t-1} \tag{103}$$

And the gradient descent for the final gate a_t:

$$\frac{\partial E_t}{\partial W_a} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial a_t}\frac{\partial a_t}{\partial z_{a,t}}\frac{\partial z_{a,t}}{\partial W_a} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot i_t \odot (1 - tanh^2(z_{a,t})) \odot x_t \tag{104}$$

With respect to U_a

$$\frac{\partial E_t}{\partial U_a} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial a_t}\frac{\partial a_t}{\partial z_{a,t}}\frac{\partial z_{a,t}}{\partial U_a} = \\ 2(h_t - y_t) \odot o_t \odot (1 - tanh^2(C_t)) \odot i_t \odot (1 - tanh^2(z_{a,t})) \odot h_{t-1} \tag{105}$$

The following partial derivatives will be defined in order to ease the process for backpropagation [17].

$$\delta h_t = \frac{\partial E}{\partial h_t} = 2(h_t - y_t) \tag{106}$$

$$\delta o_t = \frac{\partial E}{\partial o_t} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial o_t} = \delta h_t \odot tanh(C_t) \tag{107}$$

$$\delta C_t += \frac{\partial E}{\partial C_t} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial tanh(C_t)}\frac{\partial tanh(C_t)}{\partial C_t} = \\ \delta h_t \odot o_t \odot (1 - tanh^2(C_t)) \tag{108}$$

The sum is used so that the previous gradient is added to the new gradient.

Now we can define $\delta f_t, \delta i_t, \delta a_t$ and δC_{t-1}

$$\begin{bmatrix} \delta f_t = \delta C_t \odot C_{t-1} \\ \delta i_t = \delta C_t \odot a_t \\ \delta a_t = \delta C_t \odot i_t \\ \delta C_{t-1} = \delta C_t \odot f_t \end{bmatrix}.$$

And now we can define $z_{f,t}, z_{i,t}, z_{a,t}, z_{o,t}$.

$$\begin{bmatrix} \delta z_{f,t} \\ \delta z_{i,t} \\ \delta z_{a,t} \\ \delta z_{o,t} \end{bmatrix} = \begin{bmatrix} \delta f_t \odot \sigma'(z_{f,t}) \\ \delta i_t \odot \sigma'(z_{i,t}) \\ \delta a_t \odot (1 - tanh^2(z_{a,t})) \\ \delta o_t \odot \sigma'(z_{o,t}) \end{bmatrix}.$$

And finally define δW and δh_{t-1}. Recall that $z = WX^t + b$ where X^t is the input vector consisting of x_t and h_{t-1} at time t.

Then $\delta X^t = W \times \delta z^t$. And so $\delta W^T = \delta z^t \times (\mathrm{X}^t)^T$. Note here that T is the transpose. Then $\delta W = \sum_{t=1}^{T} \delta W_t$ and T here are total number of timesteps in the LSTM layer. And finally to update the weights: $W- = \delta W\alpha$. Where α is the learning rate. The outer prodtuct matrix δW:

$$\begin{bmatrix} \delta W_f & \delta U_f \\ \delta W_i & \delta U_i \\ \delta W_a & \delta U_a \\ \delta W_o & \delta U_o \end{bmatrix} = \begin{bmatrix} \delta z_f \\ \delta z_i \\ \delta z_a \\ \delta z_o \end{bmatrix} \times \begin{bmatrix} X^t \\ h^{t-1} \end{bmatrix}^T. \tag{109}$$

Notice that these delta functions are the same as in equation (98) to (105).

In order to adjust the bias, simply remove the X^T part.

$$\begin{bmatrix} \delta b_f \\ \delta b_i \\ \delta b_a \\ \delta b_o \end{bmatrix} = \begin{bmatrix} \delta z_f \\ \delta z_i \\ \delta z_a \\ \delta z_o \end{bmatrix}, \tag{110}$$

$$b- = \delta b\alpha. \tag{111}$$

3 Part 2: Implementation and results

The neural networks were made in Python 3.6 using Keras with Tensorflow as the backend. The computer used for training the models have a core i5 6500 Intel CPU and a Nvidia 750 gtx GPU.

There were two different data-sets used for this project. The 1st data-set had 900 frames of training with an additional 60 for validation, each frame is of size 14×14 as the input and 5×3 as the output, where each entry in the output represents the final fifteen checkpoints and the input represents the first 196 checkpoints. These checkpoints would register how many cars were passing through with an interval of 30 seconds for each timestep. This task was to predict 30 minutes into the future for the final 15 checkpoints. Each frame was 30 seconds totalling in 60 frames of prediction. The original input was 27×6, but an empty lane had to be added and an empty checkpoint had to be placed in the beginning so that the input could be squared. The new input have the dimension 28×7, but had to be reshaped into a matrix to 14×14. There were a total of 211 checkpoints, 7 of which were empty.

A convolutional neural network was used first and then a LSTM type network was used later to compare the results. For this attempt the overfitting phenomenon can be seen for the LSTM type network results.

The 2nd data-set had over 50000 frames, where 45000 frames were used for training. Each frame have a size of 4 as both the input and the output. This data-set was easier to predict due to the low size of the output vector (the vector we wish to predict) and the fact that this data-set had more time-steps to work with. This task was to predict all lanes at a certain time-step. We chose 24hours into the future. Each frame was a 5min interval.

For the 2nd data-set, a LSTM type network will be used only, this is because each timestep-data only is a vector of size 4. Because of the smaller input data, a CNN type network would not be needed, in general it is possible to mix CNN with LSTM, by having the convolutional layers before the LSTM layers.

For both data-sets, a validation and training graphs will be shown. This is to show the accuracy of in-data and out-data results. An error distribution will also be shown. This distribution shows the frequency of a relative error on the Y-axis and how big the error was on the X-axis. For data-set 1 a prediction of 30minutes will occur 60 times, not to be confused with 60 frames of prediction. For data-set 2 a prediction of 24 hours will occur 255 times, 24 hours of prediction equals to 288 time-steps.
The numerical relative errors calculated below is of the following formula:

$$r(a,Y) = \begin{cases} |\frac{a}{Y} - 1| & Y \neq 0 \\ 2 & Y = 0 \\ 0 & a, Y = 0 \end{cases} \tag{112}$$

Where a is the predicted value and Y the real value.

3.1 Data-set 1:

CNN:

The first attempt to predict the outcome for this data-set was to use convolutional layers. The motivation behind this type of network for this data-set was that a CNN type network will reduce the number of parameters needed to be trained. As the result shows, using CNN for time-series data is not a good idea. Despite the loss function's low value, the graph shows that taking the average (a straight line) might actually improve the results.

The blue curve is the real data, the green curve is the predicted. The X-axis represents which timestep we're predicting for a given checkpoint and the Y-axis represents the predicted and the true data. The prediction for this data-set are 60 future time-steps which is a 30 minute prediction.

Only an in-training result will be shown, as the validation graph would be even worse.

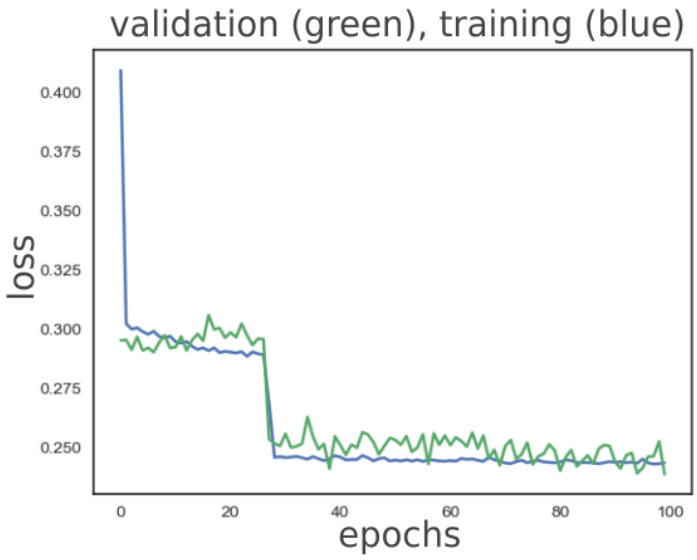


Figure 19: The loss function of the training session. Low overfitting, but also low accuracy. The loss function shows the error of all predictions vs all training data for each epoch. In this case there were 100 epochs. The blue function is the training loss and the green is the validation loss.

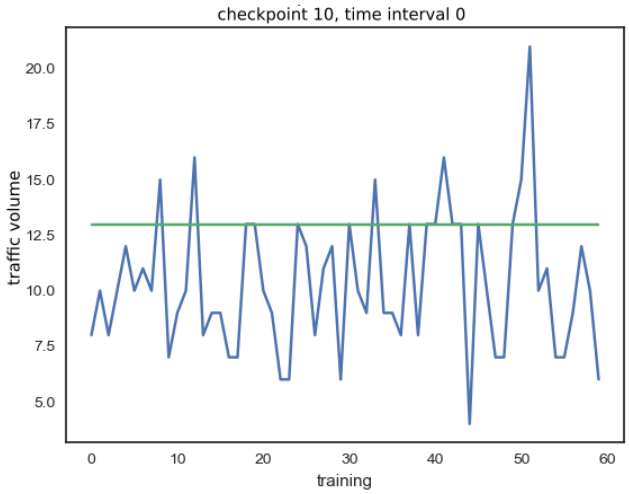


Figure 20: Checkpoint 10. Training data. The green function is the prediction and the blue function is the real data. Because of poor training results there is little sense in looking at the validation data. The validation loss function is roughly equal to that of the training loss function.

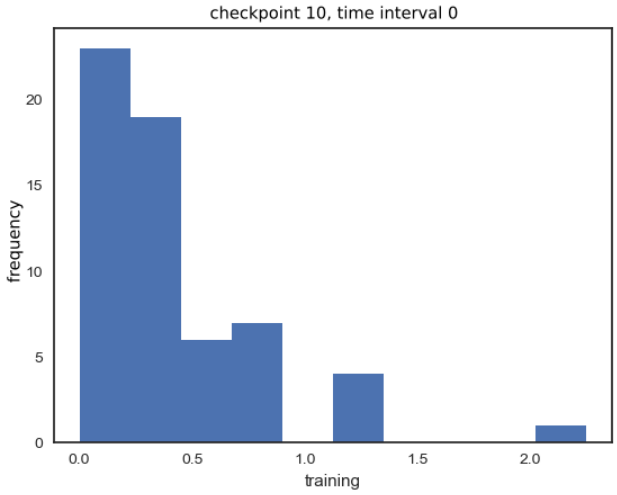


Figure 21: Training data. Histogram of the relative error of the predicted vs true data using equation 112. The network was unable to train, thus the relative errors will be high. Depending on the application, if the real data is in general uniform over time then knowing the averages of traffic volume can be good enough as a prediction.

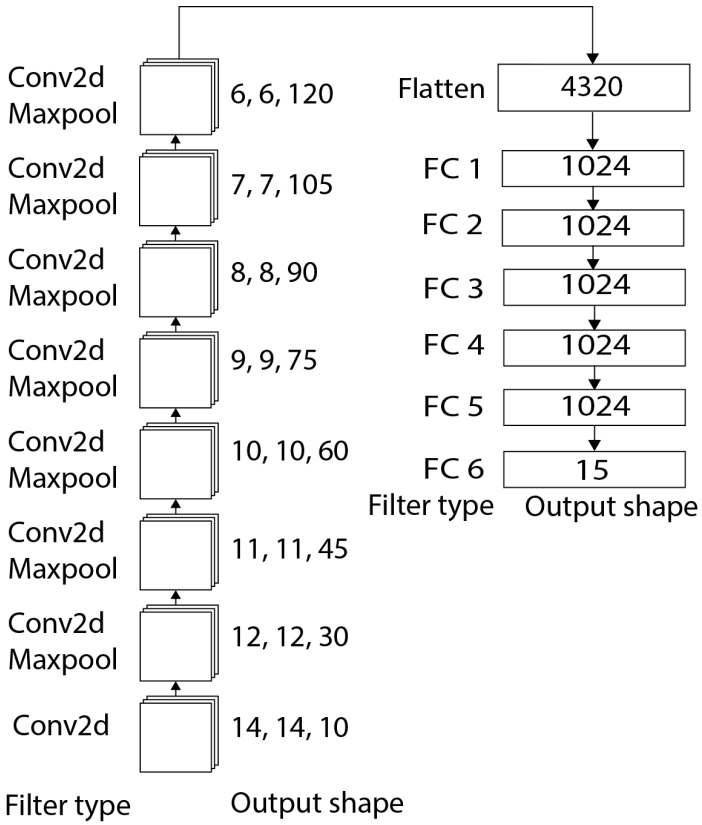


Figure 22: The network. Each convolutional layer had a dropout rate of 25% and the dense layers had a dropout rate of 20%. Each convolution increased the numbers of feature maps by 15 except for the first two which was only 10 and then 20. After each convolution a maxpool was added of size 2x2. This maxpool reduced the dimensionality by 1 (the stride was set to 1). Finally, dense layers were added which would then be compared with the true data.

Layer Type	Output Shape	Parameters	Total Parameters
Conv2D	14, 14, 10	100	100
Conv2D	14, 14, 30	4830	4 930
Max pooling	12, 12, 30		
Conv2D	12, 12, 45	21645	26 575
Max pooling	11, 11, 45		
Conv2D	11, 11, 60	24360	50 935
Max pooling	10, 10, 60		
Conv2D	10, 10, 75	40575	91 510
Max pooling	9, 9, 75		
Conv2D	9, 9, 90	27090	118 600
Max pooling	8, 8, 90		
Conv2D	8, 8, 105	85155	203 755
Max pooling	7, 7, 105		
Conv2D	7, 7, 120	113520	317 275
Max pooling	6, 6, 120		
Flatten	4320		
Dense	1024	4424704	4 741 979
Dense	1024	1049600	5 791 579
Dense	1024	1049600	6 841 179
Dense	1024	1049600	7 890 779
Dense	1024	1049600	8 940 379
Dense	1024	1049600	9 989 979
Dense (output)	15	15375	10 005 354

Figure 23: The network. Each convolutional layer had a dropout rate of 25% and the dense layers had a dropout rate of 20%. This image shows the evolution of the increasingly number of parameters. it is easy to see that the dense layer's parameters grow very fast.

Using LSTM instead of CNN:

This result show a considerable amount of overfitting despite using dropout and noise. By only looking at the loss function it is not always easy to detect overfitting. Another method to detect overfitting is to use different types of metrics. In this case the Mean Absolute Error was used. One can also detect overfitting by simply looking at the validation graphs. It is interesting, however that this attempt's validation prediction is about the same as the CNN's result.

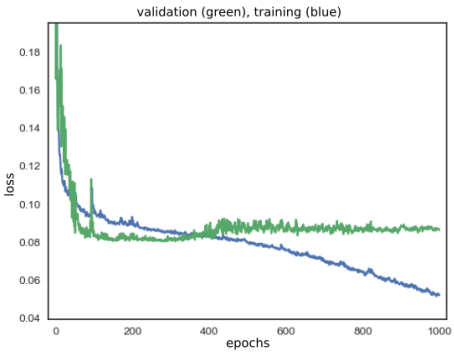


Figure 24: The loss function over 1000 epochs. The validation loss is converging while the training loss is reducing, this is an example of overfitting. Because of this overfitting, the model will do well on the training data, but poorly on the validation data. If the training continued past 1000 epochs, the model would overfit even more. Early stopping can sometimes be used to stop the network from training anymore when the validation loss and training loss starts to diverge, in this case it would be at around epoch 400, but due to the high error at 0.08 stopping early might not have been enough.

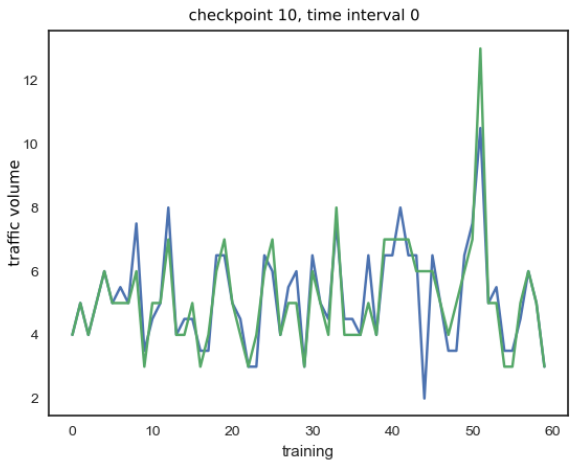


Figure 25: Checkpoint 10 of the training data. The green function are the predicted values and the blue function are real values. This model is performing well on the training data compared to the CNN model. Before the comparison between prediction and the real values, the predicted values were rounded to nearest integer, this is because the real values are all integers.

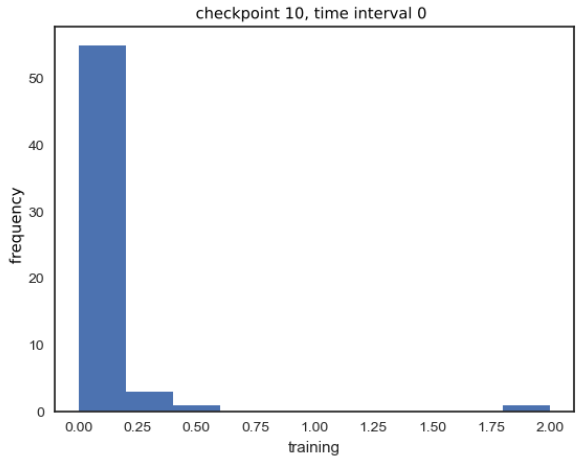


Figure 26: Histogram of the relative error between the predicted and the real values. Because of the low numbers of prediction and the true values and because the differences are on average one car difference, the resulting relative error might seem big. Depending on the application of this neural network, the method of accuracy might be relaxed for lower valued prediction sets such as these.

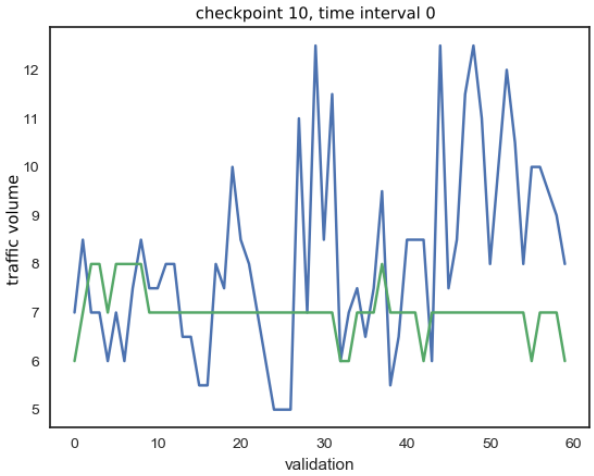


Figure 27: Validation data, same checkpoint at a different time. Note that this model was able to train a little on the validation data, observe the small spikes on the green function. Using a deeper network and a similar dataset with more timesteps could improve these results. Due to overfitting, however, the prediction is poor, similar to that of the CNN's result.

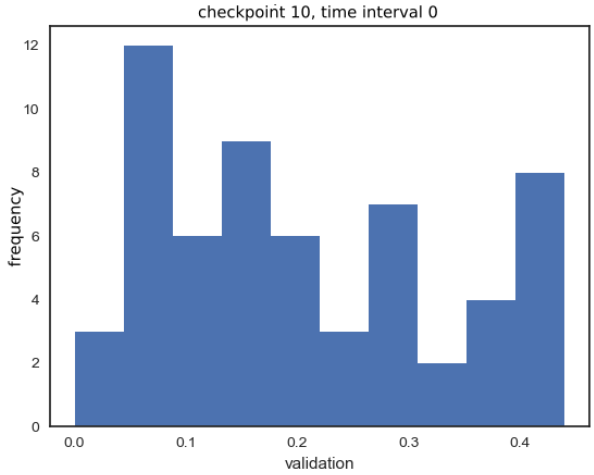


Figure 28: Histogram of the relative error. The errors are unfortunately too big for the network to be of any use. In general, having the same method of accuracy for higher valued prediction sets as for lower valued prediction sets will always favor the higher valued prediction sets. Despite this result, Figure 24 and 25 shows potential for this data-set if it had more time-steps.

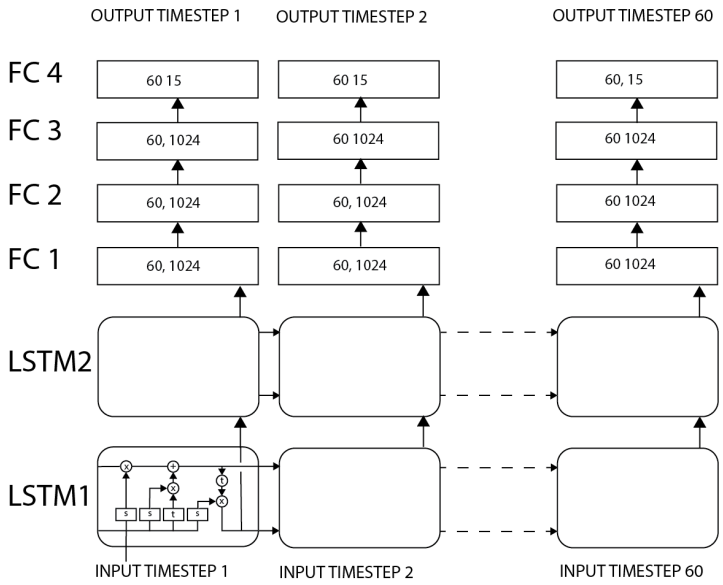


Figure 29: The network. Each layer had a dropout rate of 25%. This network is a lot simpler and less expensive than the CNN type network, despite this simplicity, the result was better if not the same. Due to the lower number of parameters, this model was faster to train than the CNN model. This network is a lot simpler and less expensive than the CNN type network, despite this simplicity, the result was marginally better if not the same.

Layer Type	Output Shape	Parameters	Total parameters
LSTM	60, 60	61 680	61 680
LSTM	60, 60	29 040	90 720
Dense	60, 1024	62 464	153 184
Dense	60, 1024	1 049 600	1 202 784
Dense	60, 1024	1 049 600	2 252 384
Dense	60, 15	15 375	2 267 759

Figure 30: The network. Each layer had a dropout rate of 25%. Compared to that of the CNN type network, this network is much simpler and was faster to train. The number of parameters is almost 1/5 of the CNN.

3.2 Data-set 2 with timestep shift:

This data-set contains over 50000 timesteps, each with a vector of size 4 which represents average traffic over a 5 minute period. The entries of the vector represents the lanes.

This network was trained on two different outputs. One with shifted timesteps and one without. Timestep shift means that for a training input X_t for timestep t, there is a desired output Y_{t+N} where N is the number of timesteps in to the future we wish to predict. The motivation behind this method is that a model will be created that predicts N future timesteps for any given input.

Just like the previous results, the X-axis shows which timestep we're predicting and the Y-axis shows the real vs the predicted data. For this data-set, the prediction was 288 timesteps in to the future which is equal to a 24hour prediction and the number of predictions for each set will be 255 which is an interval of roughly 21 hours. The first 45000 timestep entries were used for the training, 10% of these were used for validation. The final 5000 timestep entries were used for the actual validation predictions which will be presented in the results. This validation prediction spanned roughly 17 days. In short, for each day, there is a 24 hour prediction for 17 days. In total, the overall data spanned more than 170 days.

An accompanying error distribution will be shown as well. This is important because by only looking at the prediction vs the true values, it is difficult to see the average error and the accuracy of the prediction. This error graph will also give an idea of what to expect for the errors when a network like this is implemented in a real life situation.

The total accuracy is up to each individual to set up with the help of a tolerance. One way to judge the accuracy is to use the histogram images, have 90% of the bins to be under a certain error tolerance, for example 90% of the bins must be under 5% error. The error is distributed in a chi-squared fashion, by taking the area of the errors and stopping the integration at 0.05, then if 90% of the area is under 0.05, then the prediction can be deemed as accurate. Other percentage values than 90 can be used depending on the strictness. More serious applications will require a higher accuracy such as 99%.

Another attempt will be shown without timestep shifting, with the same checkpoints in order to compare the results.
For the timestep shift method, a comparison between two activation functions will be presented. One is the ReLU activation function and the other is the ELU activation function. The ELU was chosen because of its supposedly improvement over the Leaky ReLU. The activation function comparisons for the non time-shifted part will be omitted as it is redundant to show the same result twice. More specifically, the activation functions inside the LSTM layers are fixed to be the sigmoid and the tanh, however, the activation functions for the dense layers can be changed.

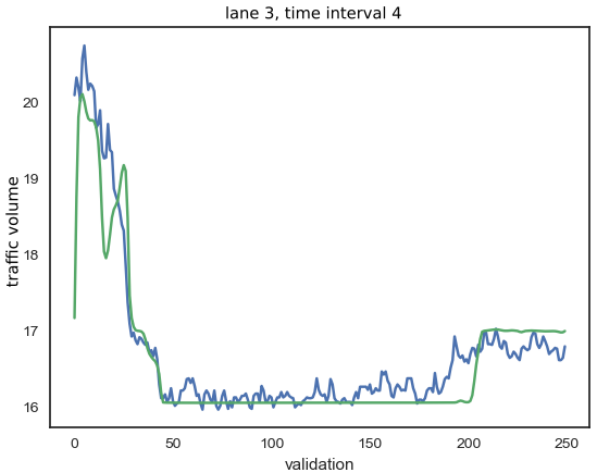


Figure 31: Validation data. Traffic volume over time (X-axis) in an interval of 255 timesteps or roughly 21 hours. The blue curve are the real values vs the green predicted values. Using the rectified linear unit resulted in good accuracy. The predictions sometimes have spikes in them which are not present in the real data, removing these can improve accuracy for some predictions, most notably at timestep 0 and timestep 25.

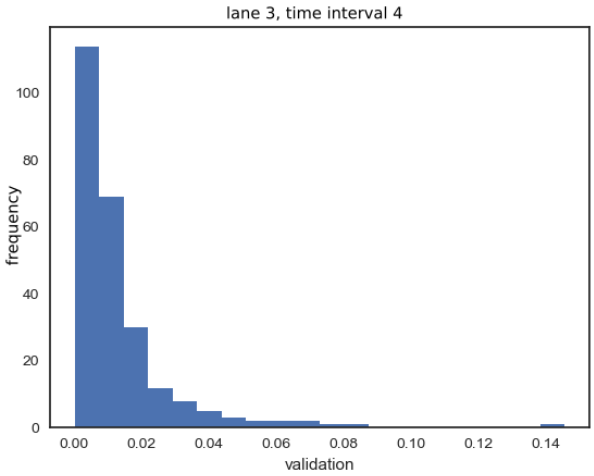


Figure 32: The distribution of the relative error between the predicted and the true values. These results were much better than the previous data-set's results. Notice the similarity between the histogram and the chi-squared function. Other functions which look similar is the exponential e^{-x} function.

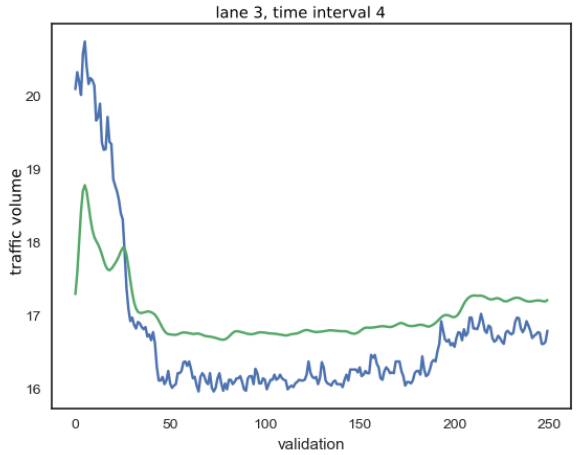


Figure 33: Validation data. Same data as above in Figure 31. Using the exponential linear unit instead of the rectified linear unit proved to be inefficient and in fact reduced the accuracy. The absolute error is anywhere between 1 and 2.

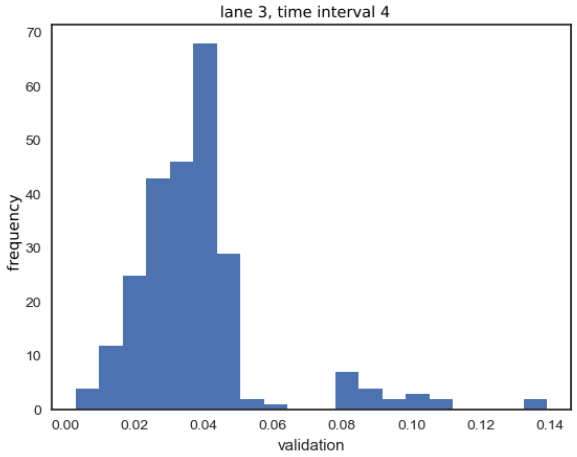


Figure 34: The distribution of the relative error between the predicted and the true values. Comparing this distribution to that of Figure 32 it can be noted that this error is greater. Despite the poorer result, roughly 92% of the error are under the 5% error threshold. As the predicted and the real values are higher in absolute numbers, the relative error becomes smaller, and this may lead to a stricter error threshold. Judging by Figure 33, a stricter error threshold should be used.

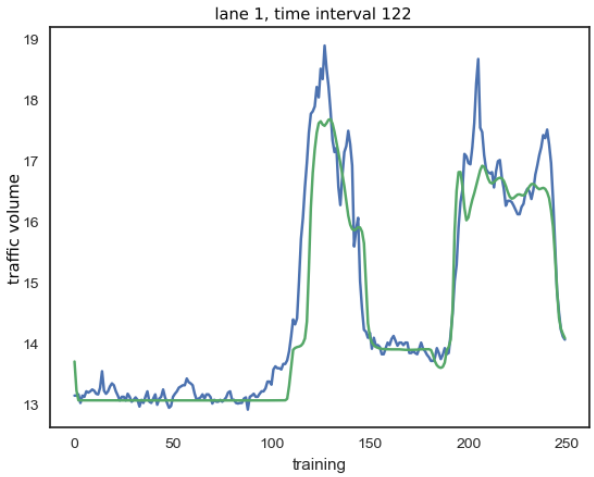


Figure 35: Training data using the rectified linear unit. It is expected that if the validation prediction shows good results, then so should the training data. Sometimes, the prediction might look a little choppy or out of place, however, looking at the error graph, the accuracy is similar to previous results in Figure 31. This graph is a good example where the real data have big spikes in them, notice this at timestep 210.

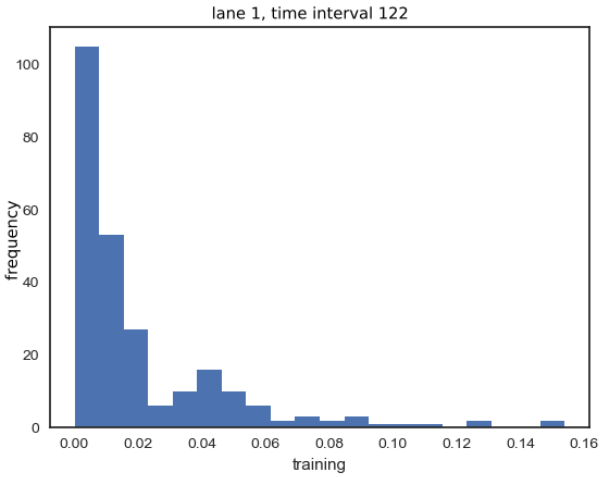


Figure 36: The distribution of the relative error between the predicted and the true values. Roughly 90% of the error are below the 5% error threshold. The error might grow where it is expected not to, this can be seen at 0.04 on the X-axis.

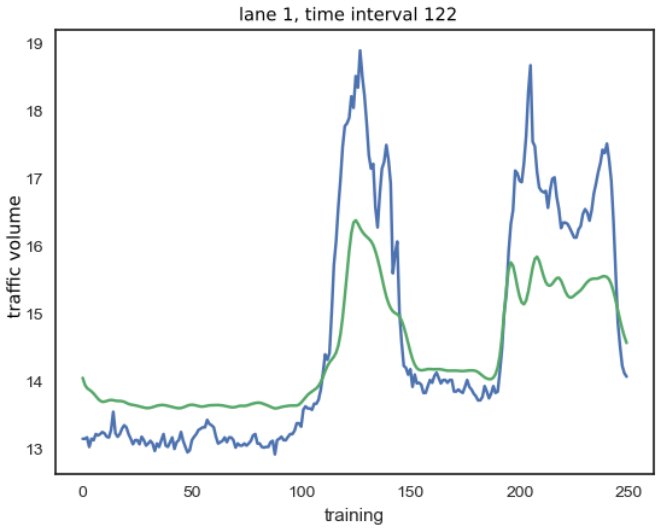


Figure 37: Training data using the exponential linear unit. Using the same real data as in Figure 35. Just as in Figure 33, the accuracy was reduced. Because the overall values of the prediction and the true values were lower than the values in Figure 33 where the values were in between 18 and 20, the relative error increased. Note that the absolute errors were about the same as in Figure 33, but because the actual values were lower, the relative error increased. This means that this method of accuracy punishes lower valued predictions and true data, while accepts higher valued predictions and true data, despite the absolute error in both predictions set is the same. Because of this, higher percent tolerance than 90 should be used for predictions set that have higher values.

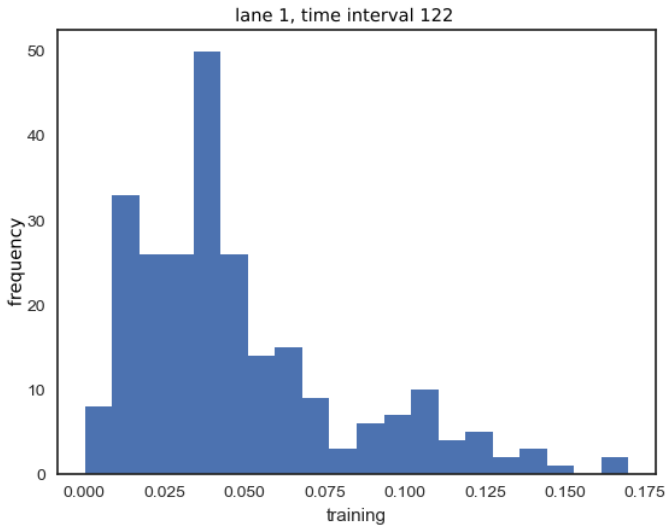


Figure 38: The distribution of the relative error between the predicted and the true values using the exponential linear unit. This time, 90% of the error were not under the 5% error threshold. If the values of both the prediction and the true data were higher in absolute numbers, then the relative error would be acceptable according to this method of accuracy used. Not only does the change in activation function change the accuracy of the prediction set, but changing to the exponential linear unit also prevents some of the prediction sets to not be accepted, thus reducing overall accuracy.

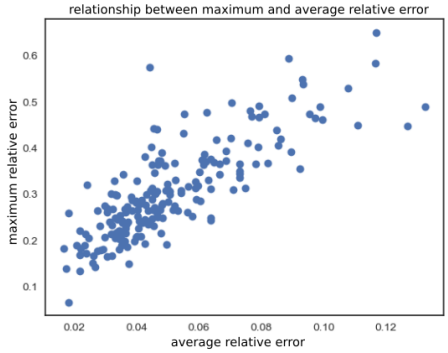


Figure 39: This plot shows the relationship between the maximum relative error (Y-axis) vs the average relative error (X-axis) for the training data. These scatter plots can also detect spikes, if the average error is small with respect to the maximum error, then there is a spike present. This can be seen for the lonely dot with average error 0.05 and max error of 0.6. As the average error increases, it is natural to assume that the maximum error will increase as well. If the spikes can be removed, the maximum error will be reduced, thus reducing the average error.

The linear regression graphs can be used to detect overfitting. If the overfitting is big, the validation dots will be more to the top-right corner with respect to the training dots. If there is little overfitting, the validation dots will be more to the bottom-left corner with respect to the training dots.

Each dot represents the maximum relative error (Y-axis) vs the average relative error (X-axis) for each prediction set. An example of a prediction set is figure (31) and figure (35).

Classification problems such as image recognition uses a confusion matrix to determine accuracy, however, for functions such as these time-series predictions, it is unreasonable to use these matrices. Instead, one way to determine accuracy is to see the spread of the relative maximum error and the relative average error. Prediction sets with higher accuracy will tend toward the bottom left and for prediction sets with lower accuracy, these will tend towards the top right.

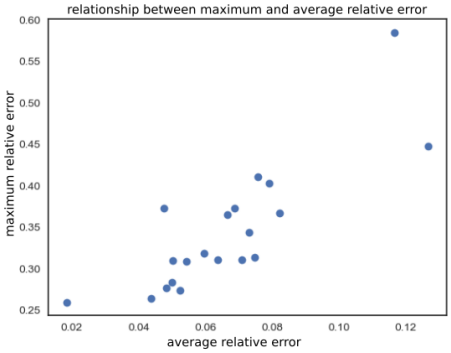


Figure 40: This plot shows the relationship between the maximum relative error (Y-axis) vs the average relative error (X-axis) for the validation data. There are a total of 20 validation prediction sets, each of these have 255 timesteps which is a total of 17 days. For each passing day, the validation predictions will have its accuracy reduced.

3.3 Data-set 2 without timestep shift:

For each input X_n a desired output Y_n for the same timestep was given. This means that the same data for the training input was used for the training output as well. The motivation behind this was to model how the traffic behaved in general independently of having the training output to be shifted in time, Using this method of supervised learning instead, it was then possible to pass in an input at time N to see the predicted result at time $N + T$. This unintuitive method of training the network would give the same results as a network with timestep shifting if not better at certain prediction sets.

The prediction sets for this method is the same in principal as before. Each function show a 24 hours prediction in a time interval of about 21 hours. This means that for a given input vector of 255 timesteps, a corresponding 24 hour prediction was made for this vector. The image below the prediction vs the true values shows the frequencies of the different relative errors. Overall these results were similar to that of the previous result with a slight difference at certain timesteps.

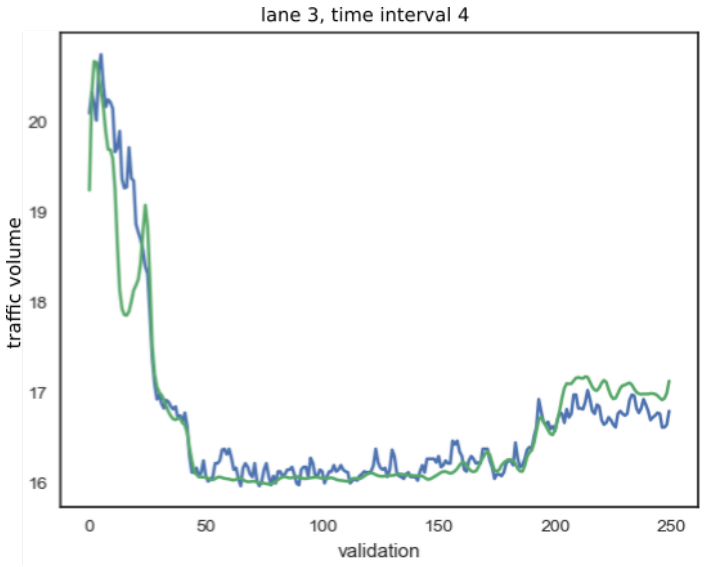


Figure 41: Validation data. This is the same true data as shown in Figure 31, this time without timestep shifting, the accuracy seems to be higher. Notice on timestep 150 to 255 of the prediction (green curve) is different from of the same timesteps in Figure 31, only this time the prediction is better.

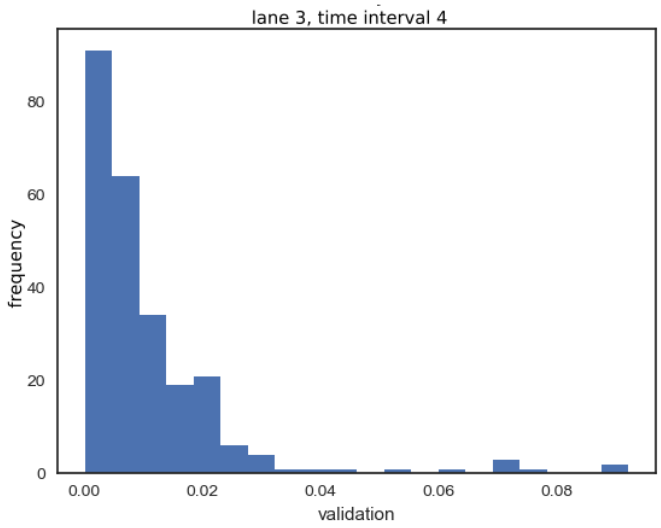


Figure 42: The distribution of the relative error between the actual and the predicted values. The spikes can be detected on the error graph by observing the bins at 0.07 and 0.09.

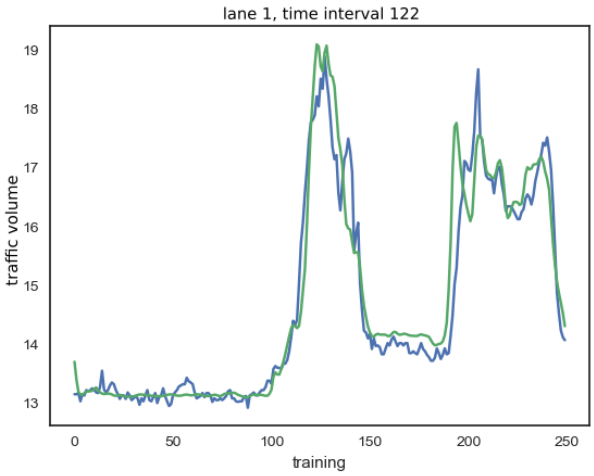


Figure 43: Training data. Comparing this prediction to the prediction in Figure 35 the overall accuracy seems to be the same. The trends in the prediction will sometimes show up earlier or later than the trends in the real data. The positive trend around timestep 100 is very close to that of the real data's trend, but the trend in the prediction around timestep 190-200 shows up too early compared to the real data's trend and this will increase error. Despite this increase in error, knowing the trend to go up can give a very strong advantage when investing money in the stock-market. It is important to judge how much of a delay (the time difference of predicted trends vs real trends) is acceptable. In this particular case, the delay lasted for no more than 10-15 minutes.

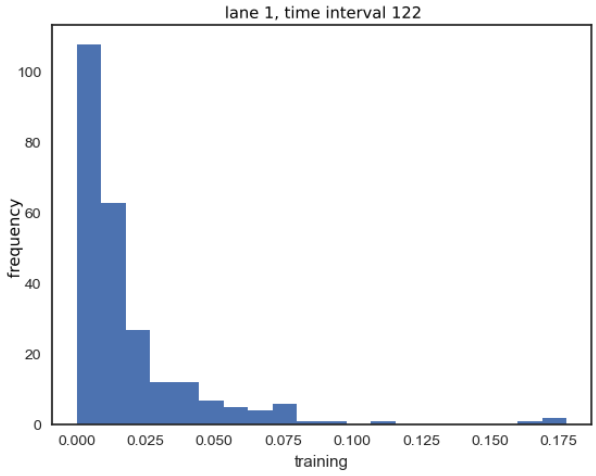


Figure 44: The distribution of the relative error between the actual and the predicted values. Using the same method for accurary, this prediction set has an accuracy of about 92%. Comparing this result to the one in Figure 36, the error did not increase as violently.

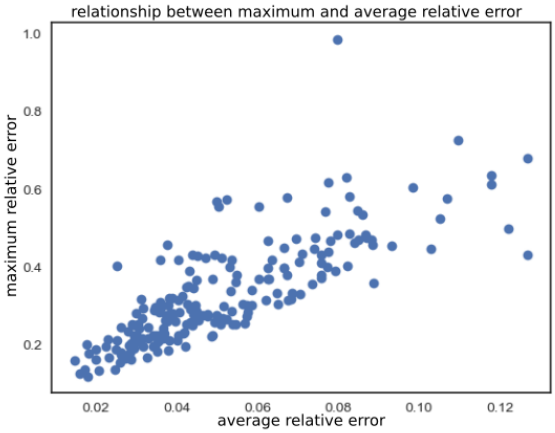


Figure 45: Training data. Y-axis shows the maximum relative error, X-axis shows the average relative error. This relationship is about the same as the relationship in Figure 39 although there seems to be more spikes present in these prediction sets than the timestep-shifted prediction sets, notably the ones with maximum relative error 0.6.

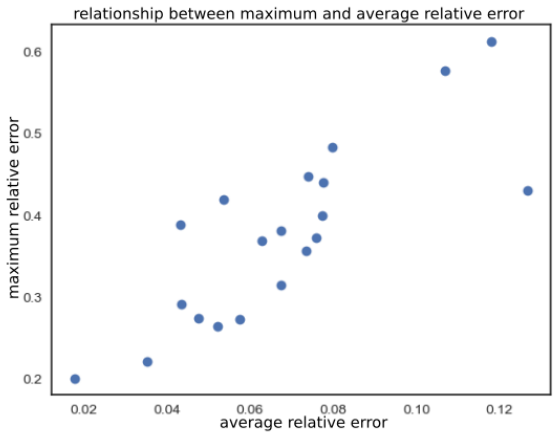


Figure 46: Validation data. The previous relationship in Figure 40 seems to be more together. Judging by this image, a small amount of overfitting seems to be present in this model, this can be seen when the dots are more spread out than the dots in Figure 40. The spikes present in Figure 45 can be reflected in this figure as well, more specifically the validation prediction sets within average relative error 0.04 and 0.08 with a maximum relative error above 0.4.

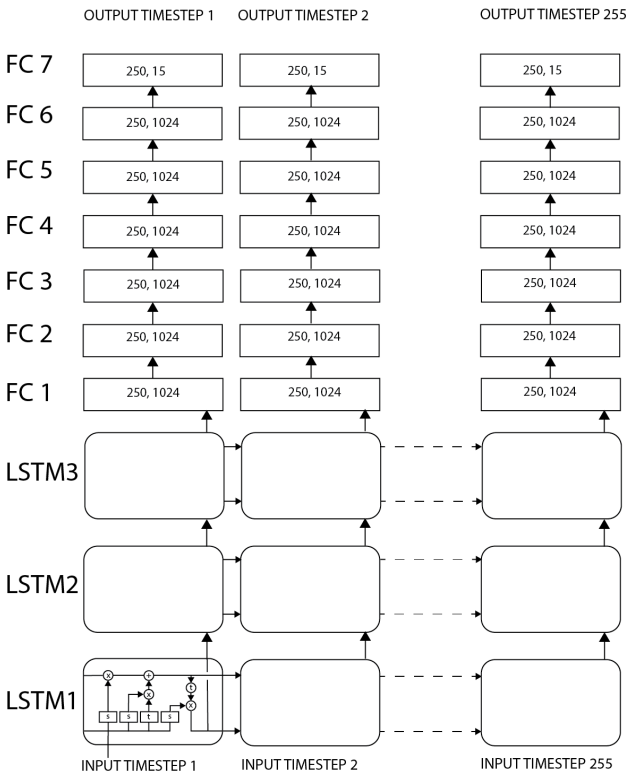


Figure 47: The network visualized.

Layer Type	Output Shape	Parameters	Total Parameters
LSTM	250, 250	255 000	255 000
LSTM	250, 250	501 000	756 000
LSTM	250, 250	501 000	1 257 000
Dense	250, 1024	257 024	1 514 024
Dense	250, 1024	1 049 600	2 563 624
Dense	250, 1024	1 049 600	3 613 224
Dense	250, 1024	1 049 600	4 662 824
Dense	250, 1024	1 049 600	5 712 424
Dense	250, 1024	1 049 600	6 762 024
Dense	250, 15	15 360	6 777 384

Figure 48: The network. Each layer had a dropout rate of 25%. This network was more complex than the previous one. The previous network was used for this data-set, but resulted in poorer prediction. Despite the number of neurons, it didn't take too long to train this network (roughly an hour and a half). This network seem to be the most successful when training on the second data-set. The shifted method of training had in total fewer errors when compared to the no-shift method of training, however, the results for the no-shift training had better trend predictions.

4 Part 3: Conclusion

4.1 The data-sets

Because the first data-set only had 961 timesteps and 900 of these were used for training, which is relatively small, the model was unable to prevent overfitting and thus out-of-training samples yielded poor results. Despite this, however, it shows potential for such a model to predict traffic by having more training data (in this case having more timesteps). The data itself was spanned over 8 hours which for real applications is unfortunately not enough. This is because it takes many days for traffic to create patterns in the dataset. The ideal data-set of this type for this project would have roughly 20000 timesteps spanning a week, but as a general principle, more is better.

For the second data-set used, results were better. Little overfitting was present and the out-of-training samples gave good results which could actually be used to predict traffic. This data-set had over 50000 time-steps which spanned several months. This type of data-set is more realistic to be used which will naturally give more robust results. The issue with this model, however, is that there were sometimes spikes in the prediction set which was not present in the real data and vice versa. In Figure 35 at timestep 200 it is possible to observe a spike in the real data, and in Figure 31 there is a spike in the prediction at timestep 10-30. Depending on the severity of these spikes, it could be advised to simply remove these spikes from the prediction by taking the averages instead.

4.2 ELU vs ReLU

The results from data-set 2 shows that not all activation functions will train the model equally. There was a clear improvement using the ReLU activation function instead of the ELU activation function despite the risk of having dead neurons. There are some speculation to this however, the traffic data (both input and output) were all positive values and as such training on negative values would be unnecessary. Using the ELU function in this case would over complicate the training session, resulting in a poorer result and perhaps increasing the training time. For a neural network where negative values are desired, the use of ELU might be a better option than that of the ReLU. For this project, only positive values were desired. Because of the similarities between ELU and Leaky ReLU, using Leaky ReLU instead of ReLU may too reduce accuracy just as the ELU did. ReLU will only give positive numbers and as a result the output for each training instance will always be positive. The fact that using ReLU was sufficient as the only activation function (outside the LSTM) can be a result of how the data was collected. The interval between each timestep was five minutes, which means that for a normal operating road, each data-point (or all four) for each timestep is independant from other data-points at different

timesteps. This means that there is no negative correlation between each data-point for timesteps closer to each other. An example of negative correlation can be when a car infront is slowing down, then the car behind must also slow down otherwise there will be a collision. Then for shorter timestep intervals (say 30 seconds) it makes more sense that the data-points between each timestep is more correlated with one and another and as a result using the ELU or Leaky ReLU can be a better idea than using only ReLU, this is because negative numbers for a neuron can be needed. To continue with this argument it is then naturally to assume that for even shorter intervals between each timestep (1 second interval) that the data-points with respect to each timestep is even more correlated. it is arbitrary to assume that for a timestep interval of 0 second will give 100% correlation because it would be the same car. As a conclusion from this, when data over time is independant from each other (with respect to the timestep) and when the data is all positive numbers, the ReLU activation function is the optimal activation function (at least when comparing with ELU and Leaky). Because tanh also maps positive numbers positively altough not linearly, it can be interesting to see the result using this activation function instead of the ReLU activation function, this however would only work for shallower neural networks because of the vanishing gradient problem.

4.3 Comparison between timestep shift vs no shift

The most intuitive method of training a time-series model with the help of supervised learning is to train with N timesteps into the future for the desired output. This will adjust the parameters in a way such that each input will have a corresponding N timestep prediction output. The downside of this model is that it can only be used for N timestep predictions and not for $\frac{N}{2}$ timestep predictions. A model like this can be used for any timestep because the prediction can be used as an input for the next prediction.

The second method to be employed was to have the desired output's timestep to be the same timestep as the input data. In other words, using the desired output as the input for each timestep. As a consequence this would lead to similar results, if not more effective at some prediction sets. When comparing the predictions in Figure 31 and Figure 43 it was interesting to see how close the accuracy were between the two prediction-sets despite the training data being so different, one method is supervised and the other is unsupervised (where the input equals the output). But it would be too early to determine the reason behind the similarities of the two learning methods. This is a phenomenon researchers are still working on to understand. It is a surprising discovery for sure and can lead to many more useful methods of training other models as well. Similarly, autoencoders are also unsupervised where the input equals the output and the motivation for this is to recreate a simpler version of the original function where the data comes from, this is used to reconstruct a lower dimension of the same dataset. The no-timestep-shift model could then predict arbitrary timesteps ahead in to the future without specifically training the network for a set time of prediction in to the future.

Both methods could predict trends (upward or downward) which can be seen at timestep 5 in Figure 31 with time-shifting and at timestep 5 in Figure 41 without time-shifting, however the method without timestep shifting had a closer trend prediction than its counterpart, notably around timestep 180-190. Comparing the upward trend around timestep 100 in Figure 35 vs Figure 43, the latter which is without timestep shifting had a closer prediction.
A property of the LSTM which can be helpful in understanding this phenomenon is that each LSTM 'block' (each timestep) can hold multiple timestep information. If each data entry for each timestep only had a single feature, then timestep without shifting would return nonsense. This means that there must be at least two or more features for each timestep in the data-set when using the non-timestep-shifted method of training. The data-set for this method in this project had four data-points at each timestep.

4.4 The neural network

The most successful network only had six million parameters, when compared to professional neural networks which have in the hundreds of millions of parameters, the network used in this project seems small, it is then expected that a network of that size will return a better result. Because of hardware limitations, the dense layers could not have more than 1024 neurons in each layer. The number of parameters in each of the dense layers would be the square number of neurons. 1024 neurons in each layer would have 1024^2 number of parameters. For a network with 2048 neurons in a dense layer, the number of parameters would be $2048^2 = 4194304$ which is 4 times than that of 1024^2. The number of paramters would explode if the number of neurons would increase.

In order to use these models, simply input a vector of size 60 for data-set 1 and a vector of size 255 for data-set 2 to then get a 30 min prediction for the first data-set and a 24 hour prediction for the 2nd. These vectors represent traffic volume for a checkpoint over time, where each entry in the vector is a new timestep. The model inputs a certain value and returns a 30min / 24 hour prediction for this checkpoint given the input value.

4.5 Judging the prediction sets

It is sometimes difficult to determine a correct accuracy when looking at the error graphs alone because these error graphs will only show the relative error between the prediction and the corresponding true value. Depending on different conditions such as for example having the absolute error to be no more than one might reject certain prediction sets which would otherwise be accepted. By taking the example of the prediction set where the ELU activation function was used (see Figure 33 , the relative error was accepted, but if the maximum absolute error allowed was set to 1, then this prediction set would have been rejected. For this reason, it might be useful to generate two different error

graphs, one graph which shows the distribution of the relative errors and a second graph which shows the distribution of the absolute difference between the prediction and the true data. If both of these errors in these graphs are considered low enough, the prediction can be used. When the values of the prediction sets increases, the relative error might be reducing, but the absolute error (the differences in absolute value) might be increasing. This can sometimes misslead the true accuracy of a prediction set.

For smaller valued data-sets with high oscillation such as the one in Figure 20, taking the average between the maximum and the minimum or taking the average over all timestep values can be useful. Because of the frequent oscillations, making accurate predictions will be difficult. Collecting the data with shorter intervals for each timestep can reduce the violent oscillation by adding more continuity to the data, however, data is typically averaged instead. Depending on the application for the data-sets, certain methods of accuracy will be more strict than others.
In the end the accuracy depends on the user's purpose, of course, being too relaxed on the accuracy will result in the network not representing future data relatively correct. Overall, the network to choose as a predictive model would be the second LSTM type network and regarding the shifted vs non-shifted method of training, the former would be used. By comparing Figure 40 with 46, the shifted method seems to outperform the non-shifted method by a small margin. Despite this, it seems that the no-shift method of training is better at predicting trends which is important when playing the stock-market. For traffic-predictions then, choosing the shifted method of training would be favourable because it would be too early to grant legitimacy to the non shifted method..

4.6 Further work

This project was about predicting time-series data or more specifically traffic data, which could also be applied to other time-series data-sets. Examples of such are predicting action outcome of a video (predicting what will happen depending on certain activities in the video) by using a combination of CNN and LSTM layers with Dense layers at the end. Action outcome predictions can be used to predict if a projectile will hit its target or if a car sliding on the road will hit another car or a person.

CNN are used to find features of an image and the LSTM layers are used because the images are time-dependant. Another example of time-series data-set which is more in connection with traffic data is the stock market. These data-sets are very similar in the way they are built. A road has several lanes where each lane can be seen as a stock. The lane's data values can be compared to that of the stock's value at that timestep. Other uses for time-series data-sets are language translation and text-to-speech synthesis which utilizes the LSTM layers. As computing power increases and the new type of computers known as quantum computers are becoming ever more advanced, such technologies will make it easier to predict the stock market and other time-series data.

References

[1] Mayank Agarwal. *CNN*. URL: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199. (accessed: 07.05.2019).

[2] Srivastava et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research 15 (2014) p 1930-1931*. URL: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf. accessed: 07.05.2019.

[3] Hichem Frigui Amine ben khalifa. *Multiple Instance Fuzzy Inference Neural Networks*. URL: https://www.researchgate.net/publication/309206911_Multiple_Instance_Fuzzy_Inference_Neural_Networks. accessed: 07.05.2019.

[4] Jason Browniee. *How to Use Metrics for Deep Learning with Keras in Python*. URL: https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/. (accessed: 07.05.2019).

[5] Amar Budhiraja. *Dropout in (Deep) Machine learning*. URL: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5. accessed: 07.05.2019.

[6] Andriy Burkov. *The Hundred-Page Machine Learning Book ch.6*. 2019.

[7] Daphne Cornelisse. *CNN*. URL: https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050. (accessed: 05.07.2019).

[8] G Cybenko. *Approximation by Superpositions of a Sigmoidal Function*. URL: %5Curl%7Bhttps://pdfs.semanticscholar.org/05ce/%5C%5Cb32839c26c8d2cb38d5529cf7720a68c3fab.pdf%7D. accessed: 07.05.2019.

[9] Paras Dahal. *SOFTMAX CROSS-ENTROPY*. URL: https://deepnotes.io/softmax-crossentropy. (accessed: 07.05.2019).

[10] Arden Dertat. *CNN*. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2. (accessed: 07.05.2019).

[11] Kunihiko Fukushima. *Neocognitron*. URL: https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf. accessed: 07.05.2019.

[12] Erik Hallstrom. *Backpropagation*. URL: https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d. (accessed: 05.07.2019).

[13] Friedman Hasite Tibshirani. *The Elements of Statistical Learning 2nd edition*. Springer, 2008.

[14] karpathy@cs.stanford.edu. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/neural-networks-1/. (accessed: 07.05.2019).

[15] A kassambara. *Linear Regression Essentials in R*. URL: http://www.sthda.com/english/articles/40-regression-analysis/165-linear-regression-essentials-in-r/. accessed: 07.05.2019.

[16] Diederik P. Kingma and Jimmy Lei Ba. *Adam: A Method for Stochastic Optimization*. URL: https://arxiv.org/pdf/1412.6980v9.pdf. (accessed: 07.05.2019).

[17] Arun Mallya. *LSTM Forward and Backward Pass*. URL: http://arunmallya.github.io/writeups/nn/lstm/index.html#/. (accessed: 07.05.2019).

[18] N/A. *Bayes Theorem in Machine Learning*. URL: https://blogs.cornell.edu/info2040/2012/10/30/bayes-theorem-in-machine-learning/. accessed: 07.05.2019.

[19] N/A. *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*. URL: https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf. accessed: 07.05.2019.

[20] N/A. *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*. URL: http://cyberneticzoo.com/mazesolvers/1951-maze-solver-minsky-edmonds-american/. accessed: 07.05.2019.

[21] N/A. *Ordinary Least Squares Linear Regression*. URL: https://www.clockbackward.com/2009/06/18/ordinary-least-squares-linear-regression-flaws-problems-and-pitfalls/. accessed: 07.05.2019.

[22] NA. *Regularization for Simplicity: L_2Regularization*. URL: https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization. (accessed: 07.05.2019).

[23] NA. *Regularization for Simplicity: L_2Regularization*. URL: https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/lambda. (accessed: 07.05.2019).

[24] Andrew Ng. *Training Softmax Classifier*. URL: https://www.youtube.com/watch?v=ue0_Ph0Pyqk. (accessed: 07.05.2019).

[25] Bharath Raj. *Data Augmentation — How to use Deep Learning when you have Limited Data—Part 2*. URL: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced. accessed: 07.05.2019.

[26] Jürgen Schmidhuber Sepp Hochreiter. *LONG SHORT-TERM MEMORY*. URL: https://web.archive.org/web/20150526132154/http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf. (accessed: 07.05.2019).

[27] S Sudhakar. *Custom Image Augmentation*. URL: https://towardsdatascience.com/image-augmentation-14a0aafd0498. accessed: 07.05.2019.

[28] Alvira Swalin. *Choosing the Right Metric for Evaluating Machine Learning Models - Part 1*. URL: https://medium.com/usf-msds/choosing-the-right-metric-for-machine-learning-models-part-1-a99d7d7414e4. (accessed: 07.05.2019).

Bachelor's Theses in Mathematical Sciences 2019:K13
ISSN 1654-6229

LUNFNA-4027-2019

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/