CE889 Neural networks and Kaggle competition

Peter Lloyd Gent  
 Chelmsford, UK  
[plgent@essex.ac.uk](mailto:plgent@essex.ac.uk)

Reg no:1802559

*Abstract -* This paper examines two possible applications of neural networks: a self driving robot that can follow a left hand wall; and a submission to kaggle for a six week prediction of shop sales. The first section focuses on a feed forward network architecture for a self driving robot and the second on a recurrent network architecture based on Longterm ShortTerm network architecture. For both cases a successful result was achieved demonstrating the applicability of neural networks to non trivial tasks.

# Introduction (*Heading 1*)

This paper introduces a solution to two problems in the domain of neural networks. The first of these is creating a neural network controller for a robot to make it behave in a left wall following manner. The second is involved with submitting a deep learning program that will forecast sales for a large number of drug stores in Germany. It is central to both of these tasks to review literature covered to prepare for the development of these solutions.

# LITERARY REVIEW

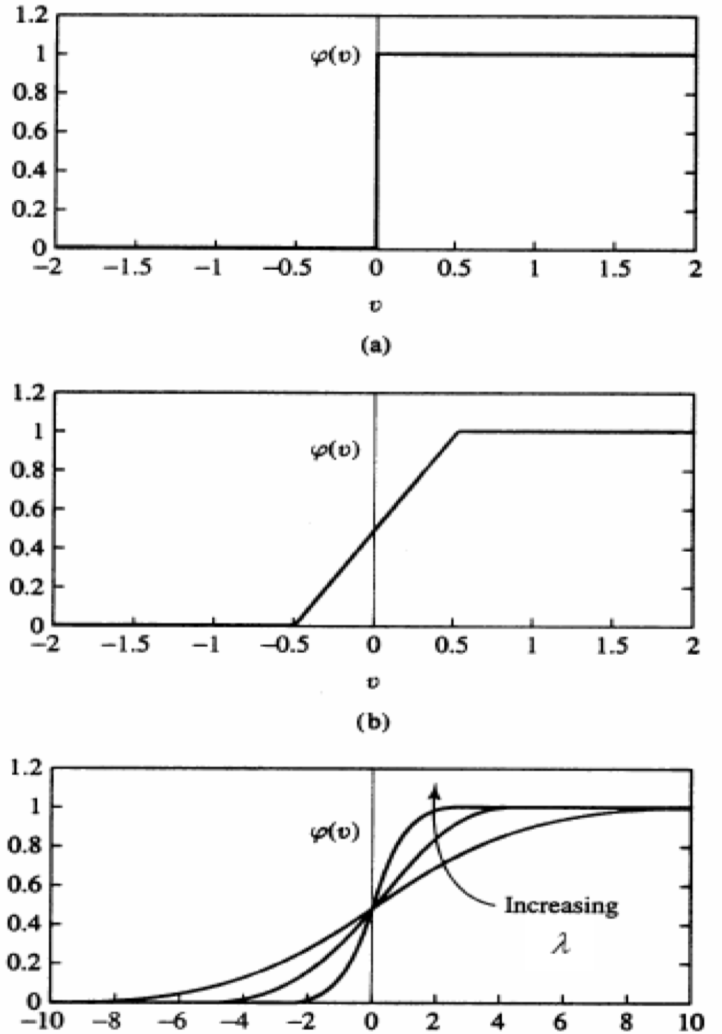
In this literary review several topics are covered. The first topic is the background theoretical material for neural networks as it is common to both the self driving robot and the Kaggle competition entry. The second is a description of the pseudocode used for the robot, as it covers much of the software architecture used for the final solution of the self driving robot. Finally, the third part of the literary review covers material specific to deep learning, including long short term memory (LSTM) neurone architecture. However the most basic question in regards to artificial neural networks is, what is a neurone?

## NEURAL NETWORK ARCHITECTURE

*fig 3. a multi layered network that is feeding forward.* [1]

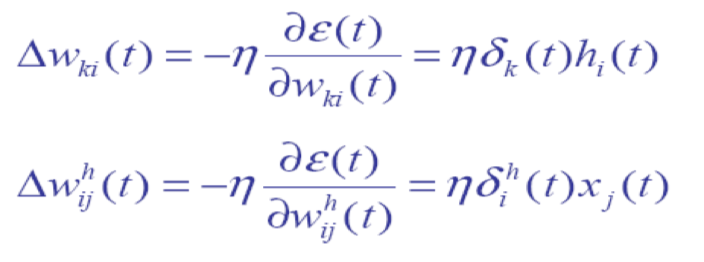
*fig 1. A single perceptron neurone* [1]

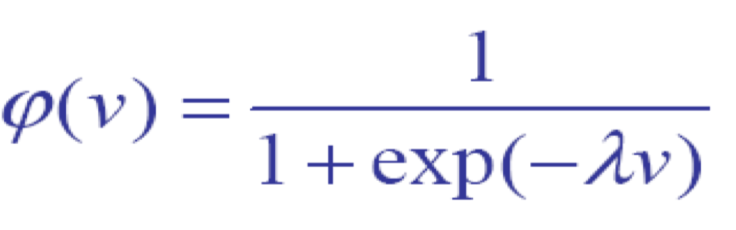
At the minimum a neurone in an artificial neurone network (ANN) comprises of a neurone, a set of weights and inputs for that neurone and an activation function or threshold function of some kind. The inputs in a single layer network come directly into the neurone itself, but in a multi player perceptron this would come into an input neurone that would feed forward into other neurones: this point will be returned to later on. The neurone sums the product of the weights and the inputs to an activation function. The activation function then decides whether to fire or not. Even in a single perceptron case, such as this it is an important design decision to choose the activation function that suits the problem at hand.



*fig 2. Different kinds of activation function (from top to bottom binary step function, linear and sigmoid.)* [1]

In a simple one neurone perceptron the binary step function would be suitable a linear regression problem, however when the problem becomes more complex different activation functions become useful for the solution e.g. sigmoid but with some caveats for each [2]. In order to examine these disadvantages the architecture of a multi layer perceptron network must be considered.

In fig 3 a 2 layer network is considered. This takes inputs to a hidden layer, and then inputs are fed into the output layers. At each stage the product of the weights for each neurone in a layer and the state are calculated, which is fed forward into the subsequent neurone. Around this point a bias weight is also normally included, as in the example. This works to shift the activation function curve as with sigmoid to the right or the left, and is usually implemented as a weight without respect to a previous layers weights[3]. The network can then be trained in order to make it useful for generalising outputs beyond the training data.

*fig5. Instantaneous gradient descent (LMS)* [4]**

*fig 6. sigmoid function.* [4]

In order to train a network from training data it is necessary to calculate the error between the current state and what is expected in order to adjust weights to an optimal value. The most popular way of doing so and training the network is gradient descent, which uses a derived error function (fig5) to calculate error with respect to change in weights. The idea of gradient descent is to find the point where the error is minimised i.e zero, at which point all of the weights are optimised [5]. In the equation the large N represents the learning rate, which acts as coefficient to determine how big a step the gradient descent algorithm takes to adjust the weights until the error is 0. If the learning rate is too large it will step over the optimal weights value, and if it’s too small it will take too long to train [5]. Finding the goldilocks learning rate is therefore as a key part of the design process for creating a gradient descent with momentum neural network.

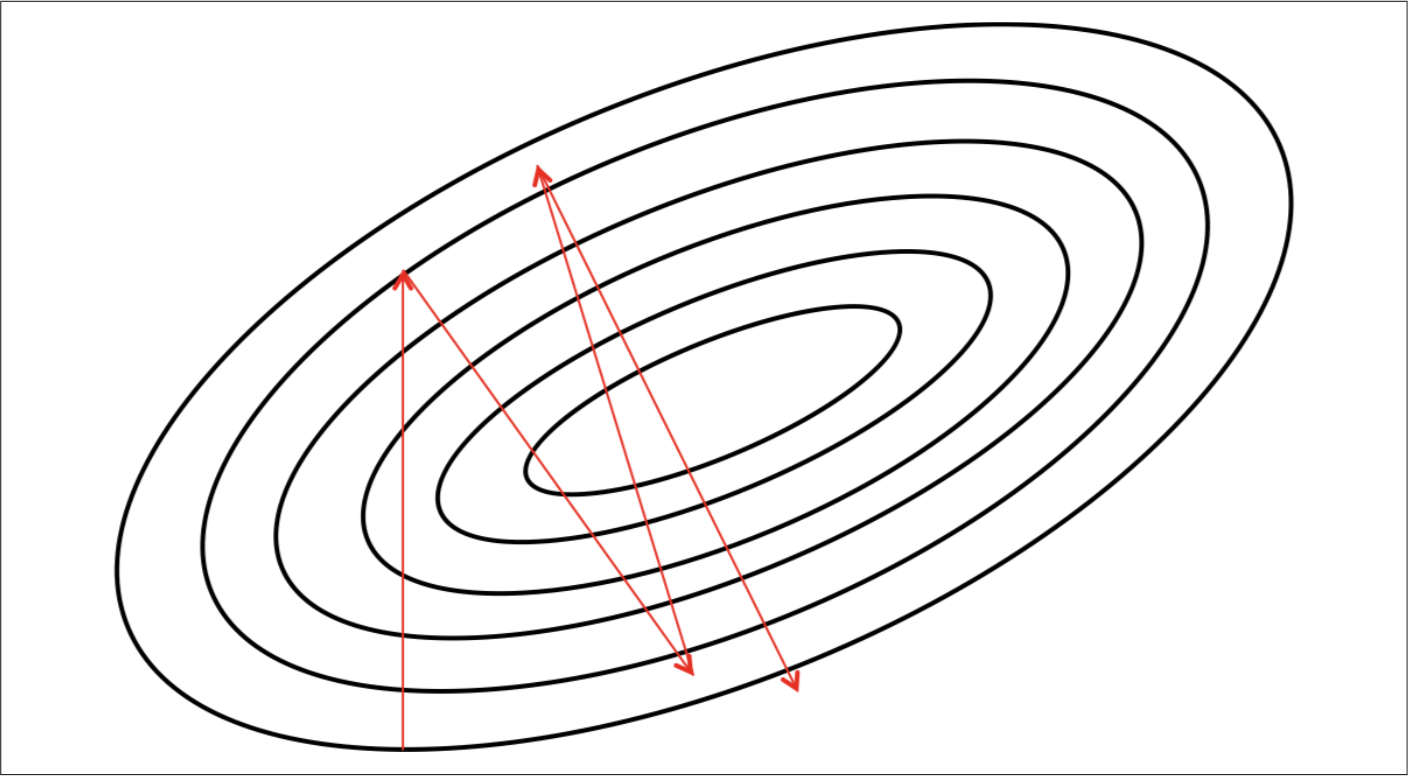


Fig 7: Stochastic gradient descent visualisation with a small learning rate [7]

A stochastic gradient descent with momentum network is preferable to a stochastic gradient descent (SGD), which can be visualised above as a series of contours created from a loss function. The main problem with SGD is that learning can be slower than expected because of noise it can take a

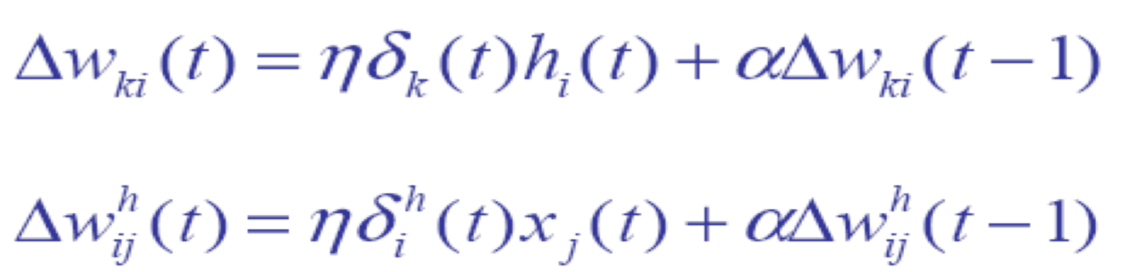


Figure 8: Gradient descent with momentum [4]

very long winded approach to reaching minima or the optimal loss amount of 0 (if at all). Asides from changing the learning rate there is not much that can be done except for considering an alternative form of gradient descent: stochastic gradient descent with momentum (SGDM) [8]. A SGDM approach means that momentum is added to the algorithm making the algorithm perform in the same way as a ball accelerating with gravity pulling a ball to the bottom of a dip. This form of gradient descent is particularly useful with stochastic gradients (the other option being batch gradient descent that processes an entire batch at a time to get the loss function) where each value is examined one at a time. The random nature of this means that there may be more noise with the gradients and so the momentum effectively pulls the solution towards 0 quicker than standard SGM. For this there is a Alpha hyper-parameter that works on the prior or previous seen weights to increase the pull towards 0 at a constant rate (see fig 8).

Other design choices that are just as important are the lambda to be used for the activation function (derived from fig 6), or choosing enough neurones to not under-fit or overfit the solution [4] and what activation function to use. In all cases it is a matter of trial and error to design a neural network of the right size, with the right number of layers and the right parameters. For the activation function a major disadvantage of a linear activation function is that linear activation functions have a constant gradient [2]. This makes gradient descent impossible as the gradient will always be a constant. Alternatively, by using a sigmoid function you can get the best of both worlds between linear and step. However, using sigma can be problematic as it will fail to learn quickly or at all at the sharpest points of the curve (trending towards 1 or 0) [2]. It is for this reason that a sigma activation function (fig 5) was chosen for the self driving robot.

*PSEUDOCODE*

The solution for the self driving robot is based on pseudocode from the book Games Artificial intelligence by Millington et al [8]. As this gives insight into the general structure of the solution it is described in depth here. Firstly the network class contains three lists, one for each layer of the multi layered perceptron class. It also contains a function generateOutput to initialise the input layer and iterate through and feed forward the perceptrons from the perceptron class for the hidden and output layers. The network class also contains most of the back propagation code for the solution. It does so by iterating through the output neurones, calculating the error for each one and adjusting weights. It then works through the hidden perceptrons, correlating the output state to each of the neurones and then calculating the error from both output neurones to adjust the weights in the Perceptron class. The Perceptron class contains a vector of weights that corresponds to a vector of input perceptrons. Each perceptron holds the state and error for that neurone. Finally, there are three functions in the perceptron class: feedforward; adjust weights; and getIncomming weight.

The function feedforward is called from the Network class. It sums together the product of weights and states to feed forward to set the state with a thresholding or 'activation function’ of the kind sigma. The function adjust weights on the other hand takes error from the back propagation and then uses the learning rate to scale the amount of error against the state. This is added to the input weight to get a new weight balanced in respective to the error for a given neurone. It then stores the error. Finally, the getIncomming function takes a perceptron neurone from the networks class in the process of performing back propagation. It looks for a corresponding perceptron in the input perceptron list for a given perceptron and returns that weight.

*LSTMs:*

There are many kinds of RNN that exist: LSTM and gated recurrent unit (GRU) are two of these. These two architectures overcome the vanishing gradient problem that is common to RNNs, which leads to untrainable networks as more and more layers are added to a neural network [8]. In the simplest terms RNN architectures work by processing a series of results (possibly time series for example) and retaining an internal state of the information relative to what has been seen before [10]. In effect an RNN works like a reader processing each word at a time while retaining the understanding of the entire sentence. It is common to all recurrent neural networks to have an internal loop instead of just feeding forward in the architecture.



*fig 8. LSTM and GRU architectures* [11]

The LSTM architecture (above) is detailed by Shen [10], it gets over the problem of vanishing gradient with input, forget and output gates. Instead of neurones the LSTM solution and GRU solutions are memory blocks that are stacked together to process sequences of data as layers. The gates operate as follows: the input gate regulates how much of a new cell to keep state of, the forget gate regulates how much of the existing memory to forget and the output gate regulates how much of the state should be exposed to the next layers of the network. Thus the entire cell can be considered to be like an advanced activation function with all of the gates into and out of the cell and the state inside the core of the cell. On the other hand the GRU architecture presents a simpler approach of operating a reset gate and an update gate. It does so in order to either forget a previous state (reset) or to decide on how much to update at a time (update)[12]. For the purpose of the assignment a LSTM approach was selected because the documentation was better for LSTMs in Keras than for GRUs.



*fig 9. Stacking together of many layers of LSTM* [9]

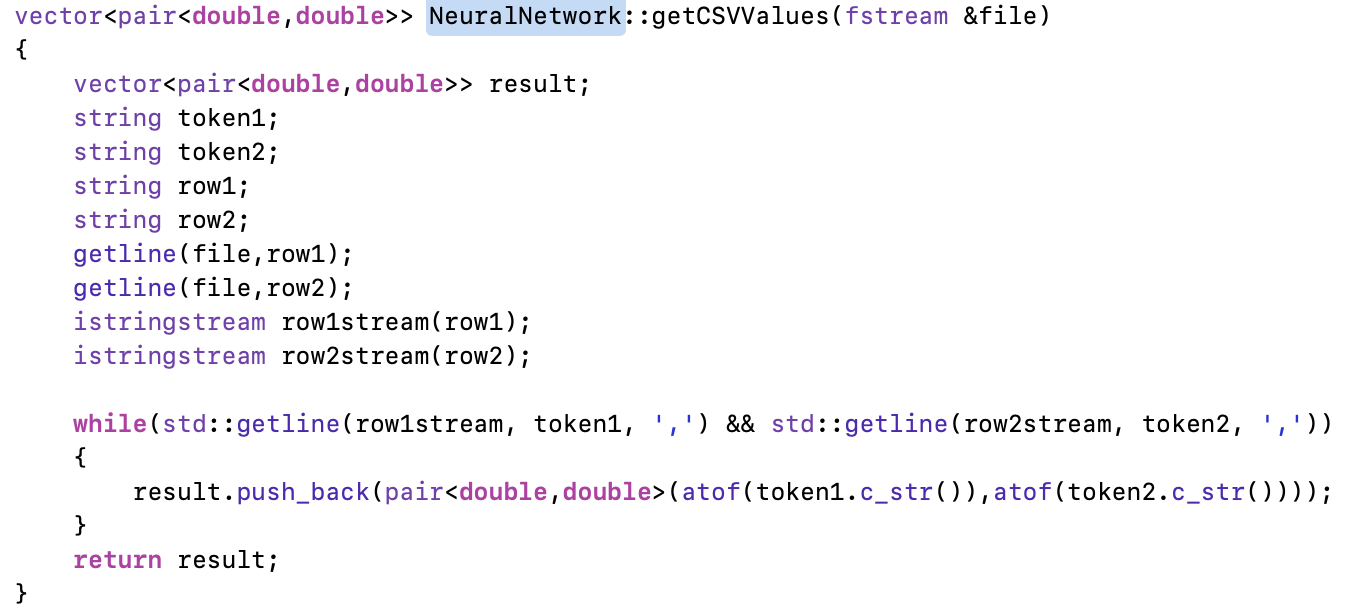
In a multi layered LSTM network the basic principles that apply to an individual LSTM cells are carried over to a multi layered LSTM network and is described in detail by Chollet [10]. The input, output and forget gates now work together to either retain information in the carry track (the storage of what has come previously for each cell) or forget about some data in the carry track. In fig 8 the cell t-1 it starts with a blank slate that computes what to carry over state into the next cell in the sequence. At the point of processing input for the next cell in the sequence takes the state from the previous cell and then carries over information from the big picture of whatever has become before (CT). As described previously each interval the cell decides whether to forget some of the irrelevant information that the cell holds by multiplying what is in the carry track. This is done by multiplying a weight f\_t against the information given from the carry state. At the same point the weights of the input and the state are multiplied to potentially take in more information. Any information that is not inputted is potentially included in the memory for a second pass of the carry computation. For the purposes of the Kaggle competition the solution that has been developed uses supervised learning instead of unsupervised learning as the approach. This means that the multi cell network has to be trained effectively through an applicable back propagation algorithm. The algorithm selected for this purpose is back propagation through time (BPTT).

In a supervised recurrent network such as an LSTM it is necessary to use an algorithm like back propagation through time (BPTT) to train the network in a similar manner to a feed forward network. What BPTT offers beyond vanilla back propagation is a consideration for time lags. In effect for BPPT diverges from back population by unzipping a time step sequence of outputs that occur sequentially one after the other. Thus, the network is shown one output after the other in the order that they occurred time wise. This leads to the network being rolled back and then then calculating the error before rolling the network back again to update the weights [13]. By having memory of what has happened before a LSTM BPPT network is particularly adept at application such as motion video, speech or in the case of this paper a Kaggle competition to forecast shop sales.

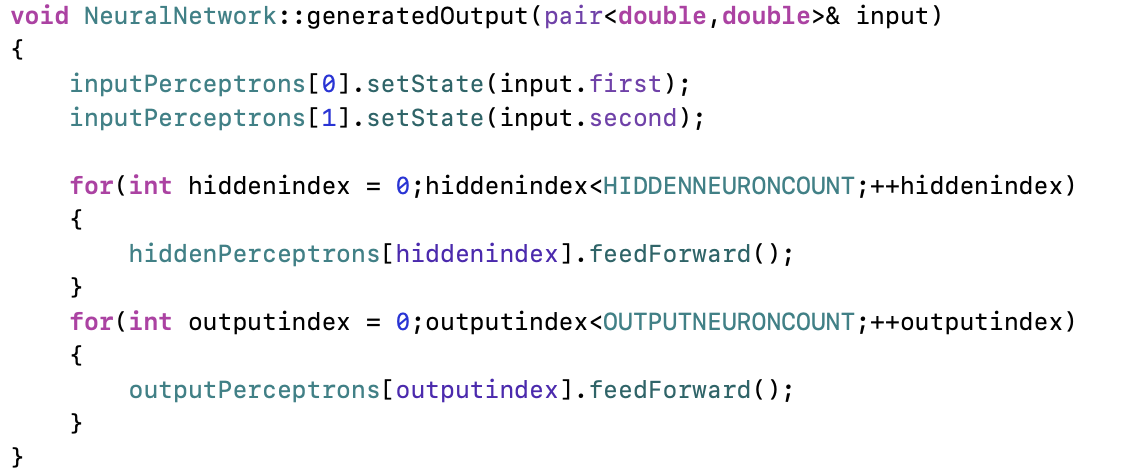
# Neural networks robot implementation

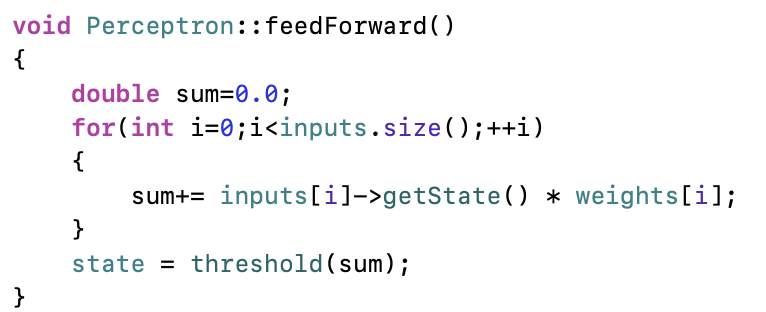
The first iteration of developing a neural network solution for the self driving or left wall following robot was to try to code without Matlab as a starting solution for the left wall following robot. In order to develop a solution the following experimental stages were followed: collecting data; creating the network; configuring the network; Initializing network weights and biases; Training the network; validating the network and using the network. For gathering data a PID solution that followed the left hand wall was used to gather data for motor speeds and ranges for left and left forward sensors. The experimental procedure was for this iteration to simply use trial and error in the MobileSim simulator and in visual studio to prototype the solution bit by bit.

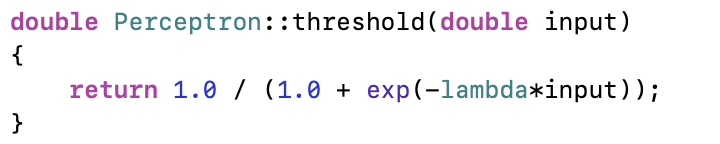
With the first implementation the program took modified data exported from matlab. This was taken in the format of a transposed csv file that had the duplicates removed in excel and shuffled in matlab or excel and were normalised according to the max wheel speed(300) or the max range for the sonar sensor (5000).



In order to take data in the program uses a helper function (above) to get a line each time, tokenising the comma separated value (CSV) file into strings, which in turn got converted into double format for each pair of results (above) The result is a vector of pairs for left and front ranges and left and right motor speed separately. The goal of training the network is therefore to correlate changes in range with changes in wheel speeds so that the self driving robot follows a left wall. This would be impossible without functions for training the network.

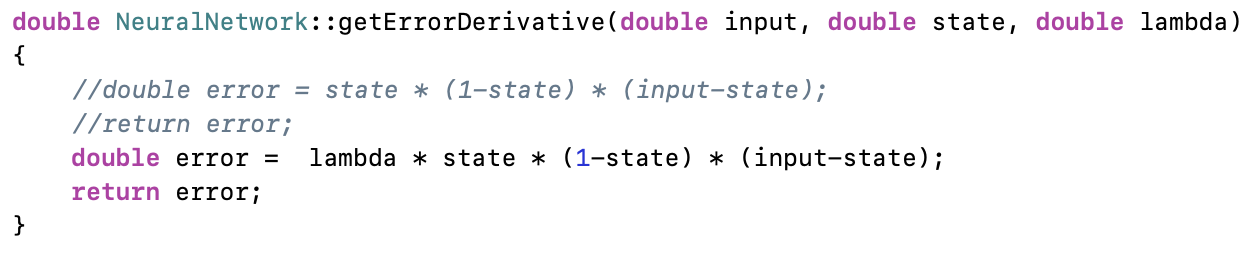


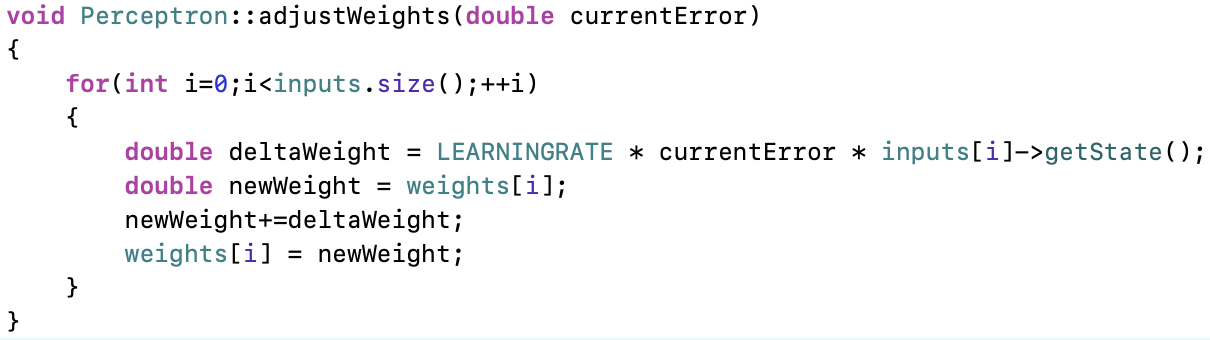




The next operation that the code performs is to learn the data set to create a weighted network that can generalise from the training data left and right wheel speeds. The code above demonstrates the first phase of this: feed forwarding. In this function the input neurones are initialised with one of the sonar ranges. Then each of the hidden and output neurones are made to call the feedForward function (above), which sums together the weights and the state of each input neurone to the called neurone: stored in a vector of Perceptron pointers. Finally at the end of feeding forward for a single neurone the threshold or activation function is called to determine the firing strength for the neurone. At this point it’s worth noting that in the first iteration (upon demonstration) a lambda constant was not a part of the code, this was added later on to allow for fine tuning the behaviour of the activation function. Another late feature was also the inclusion of a bias, which effectively can shift the activation function left or right in the case of a sigma activation function. This was introduced to try to see if bias would make the network train, but efforts in this direction did not result in a network that would train properly.







The next step after feeding the inputs through the network is to work out and then back propagate the error from the output neurones onwards to the hidden neurones. In the code above this starts by declaring two vectors, one to hold each of the expected output (motor speeds) and another to hold the error to calculate the gradient later on. In the first loop the code gets a perceptron from the list of perceptrons detailed in the literary review. It then gets the state and calculates the error using the function getErrorDerivative (see above). In the first incantation of the function a lambda was not included, this was included later on to be able to fine tune the error calculations. Finally for the output neurones the weights are adjusted for each of the output neurones with the error calculated from the derivative. A similar process also occurs for the hidden neurones whereby each of the output neurones get a respective hidden neurones weight which is then adjusted according to the error for each output neurone.

The result from a first iteration of the code was very basic. It did not feature the calculation of mean square error after each epoch, the re was also no lambda in the sigmoid thresholding or activation function and the network would not train correctly as the network would not gravitate towards 0 error. The robot presented itself as just moving aimlessly in a circle or with more neurones as a robot that would drive straight ahead regardless of obstacles or changes to the wall to the left of the robot. At first it was believed that the data might be at fault so the process of creating a training data set was reviewed and reviewed again to make sure that the data was not at fault. In the end moving towards a second iteration of the neural network involved both looking at the data in the data set and also at the code itself.

The second iteration improved beyond this by returning to the drawing board and re-examining both the code, the data going into the solution and the methodology used, specifically: the amount of testing in matlab and the use of hyper parameters for it. The first task was to create a mean square error output so that on each epoch the overall amount of error for that giving training session could be reported to give some kind of metric for

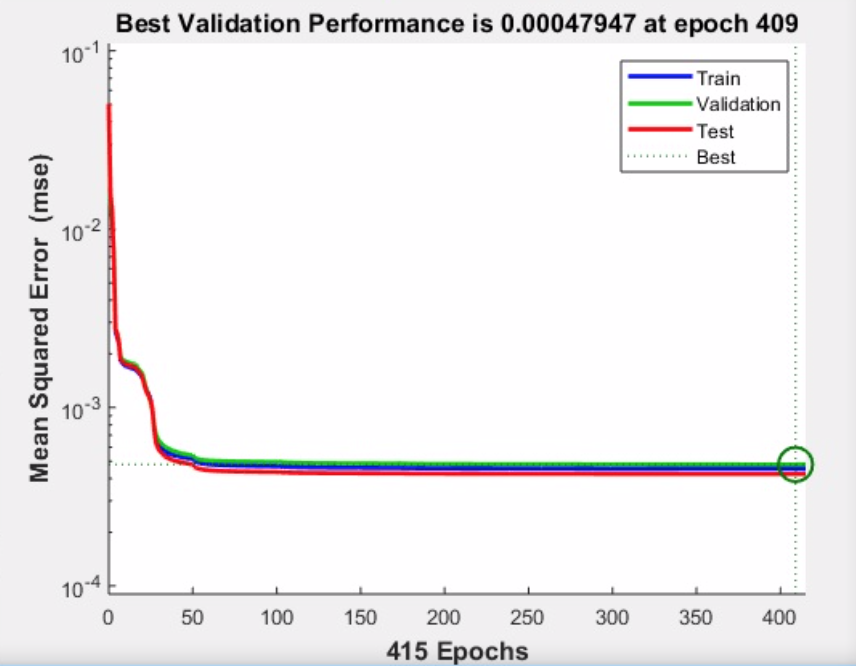


Figure 10: test with four neurones over 409 epochs

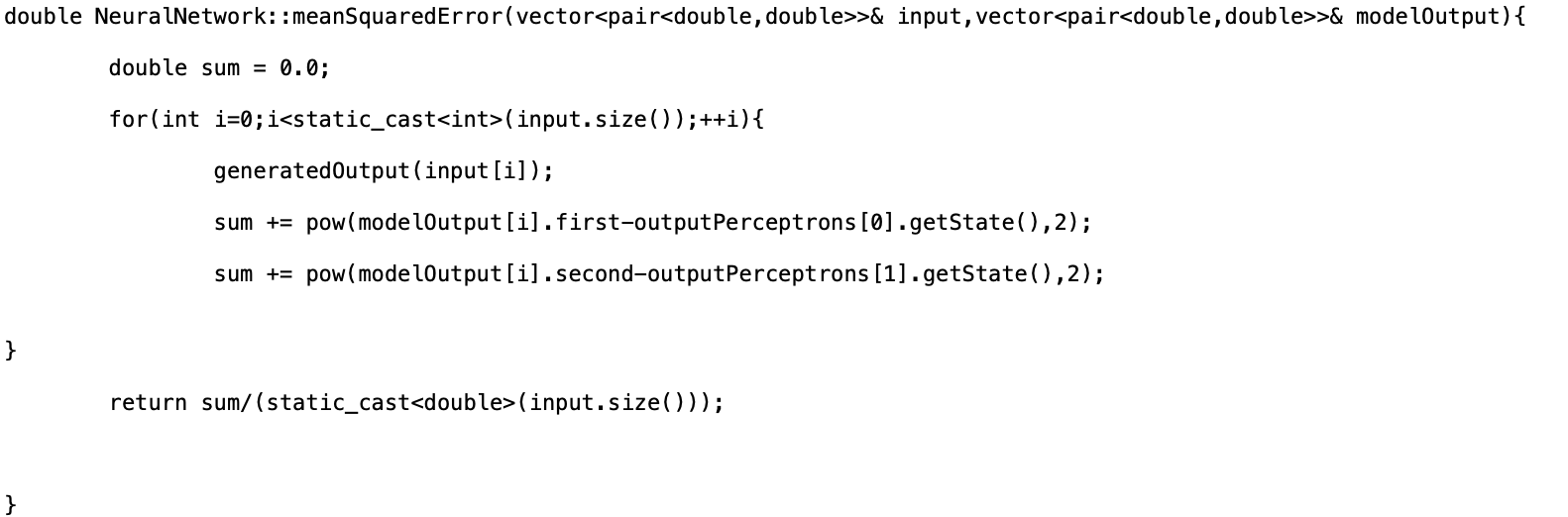
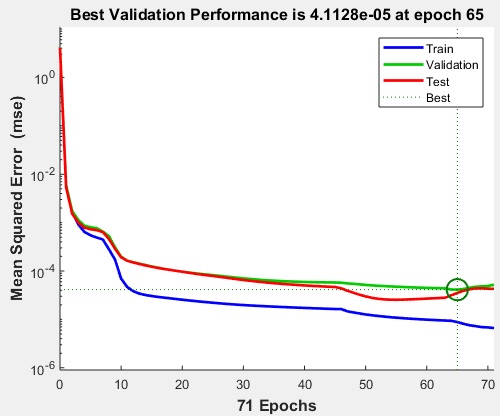
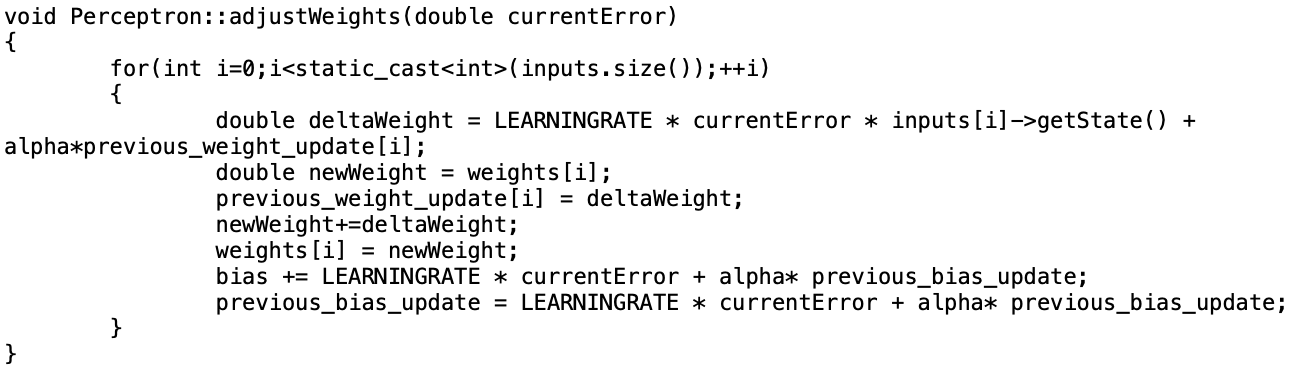


Figure 11: test with one hundred and fifty neurones in matlab.

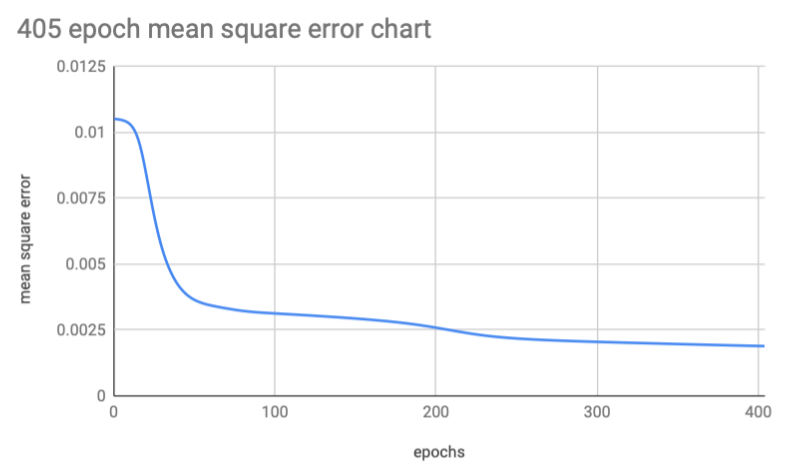
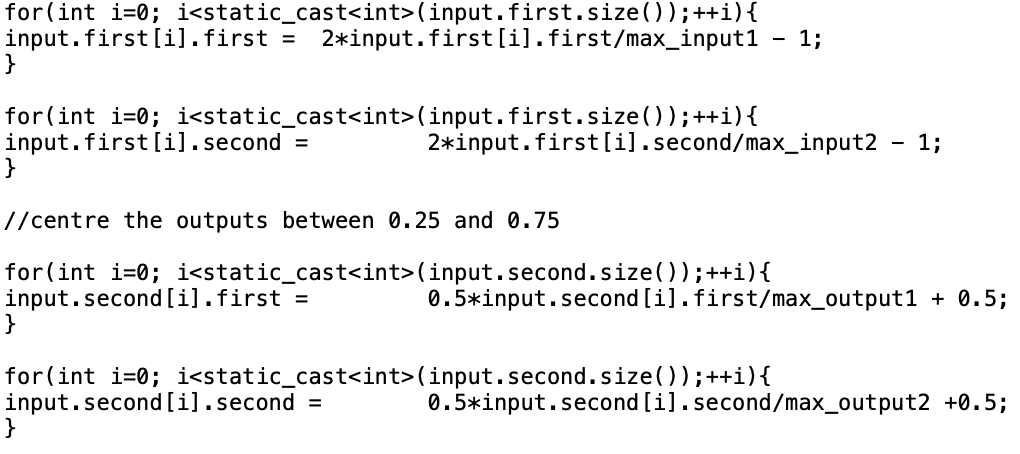


Figure 12. results of solution with 405 epochs for solution with SGDM and bias.

As can be seen from the above graphs the end result of the second iterations addition of features such as bias, and SGDM is comparable to that in matlab for the same number of neurones (fig 10) where the results trend towards being asymptotic. In order to test the fitting of the solution further and make sure an appropriate number of neurones were used an extreme number of neurones was also tested, using 150 neurones. This proved to provide much worse performance albeit with some similarities in the shape of the graph output for 4 neurones. A vital aspect of this process (beyond verifying their performance) of running experiments has been to work out values for learning rate, Alpha (SGDM), Lambda (activation functions). The use of matlab helped with this for learning rate in particular, and with a process of trial and error with matlab and the program itself a learning rate of 0.06 was set. In the end no Lambda was set for the activation function (set as a constant - 1), but an Alpha value of 0.25 was set to scale the velocity of weight adjustment. In order to put these variables to use a number of helper functions were created.

One of the first elements that was added to the second iteration were functions such as remove duplicates, randomise and preprocess cv to do all of the data processing in a standardised format. The output from this leads to a function extractFromStandardCsv that gets the lists of ranges and motor speeds and separates both into a vector of pairs for motor speed and ranges respectively.



The next major function PreProcessCv (above) centres values between -1 and 1 by getting the maximum first of input and outputs. It then multiplies by a constant value and divides by the max to centre between 0.25 and 0.75 or -1 and 1 which better models output to the sigma activation function. It is later on that it is decentralised for motor speed output by performing the actions in reverse. Also included in the second iteration were functions for reading or writing weights, calculating mean square error, setting and getting bias and gradient descent with momentum.

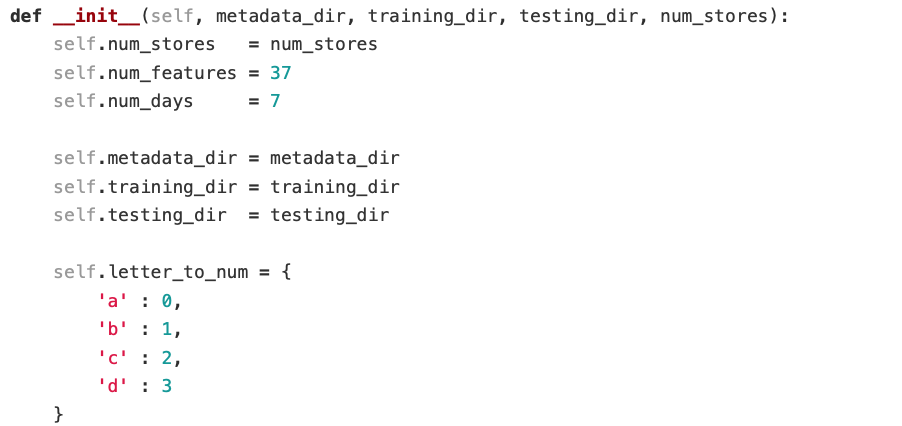
The introduction of a bias and mean square error (MSE - above) allowed for performance comparisons between different epochs. It takes the sum of errors for the left and right hand sides and adds them to a sum that is divided by the number of entries. The performance was then able to be declared for things like having bias and not bias in a timely manner because of the implementation of SGDM (above) to go with MSE. The main additive to the code is an alpha multiplier to previous weights to carry over momentum from previous cases. In cases before the implementation of SGDM it would take many hours to get the performance characteristics of bias and non bias cases. It is thanks to that efforts like this a noticeable performance gain with bias could even be compared or evaluated later on, as has been done.

# KAGGLE COMPETITION

The main aim of the Kaggle competition is to predict the sales for 1115 shops across Germany. The dataset provided for the task includes a multivariate feature set, including things like promotions, competition, school and state holiday, seasonality and locality [13]. What is expected is to be able to predict accurately the sales figures given the metadata given for promotions, holidays etc for up to six weeks. The challenges that arise from such a competition are numerous. Firstly, it is important to consider what features should be included in the model for the solution. It could include all of them or only some of them. In examining the data further the major challenge is that of shop closures.

The special case of shop closures in the competition data was is a critical part of the solution to write in. The most basic assumption from this point is that shop closures mean that all sales data is 0 for that day - this is pronounced on Sundays where a majority of shops were closed. The way in which this was tackled was to average the sales data before and after the closed period, and in prediction to set output as 0 for the day a shop is closed. Asides from shop closures all sales data can infer things like state\_holidays as a regular pattern in the data. These patterns in the data occur are retrieved from the process of training in the LSTM class.

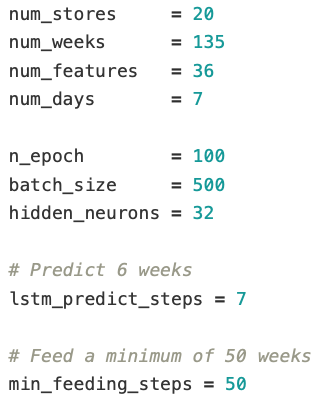
The solution for the Kaggle problem is divided into two py files: DataFolding.py and LSTM.py. As the name suggests the first of these files is responsible for folding the data to be cross checked and the second is responsible for the training and operation of the network.

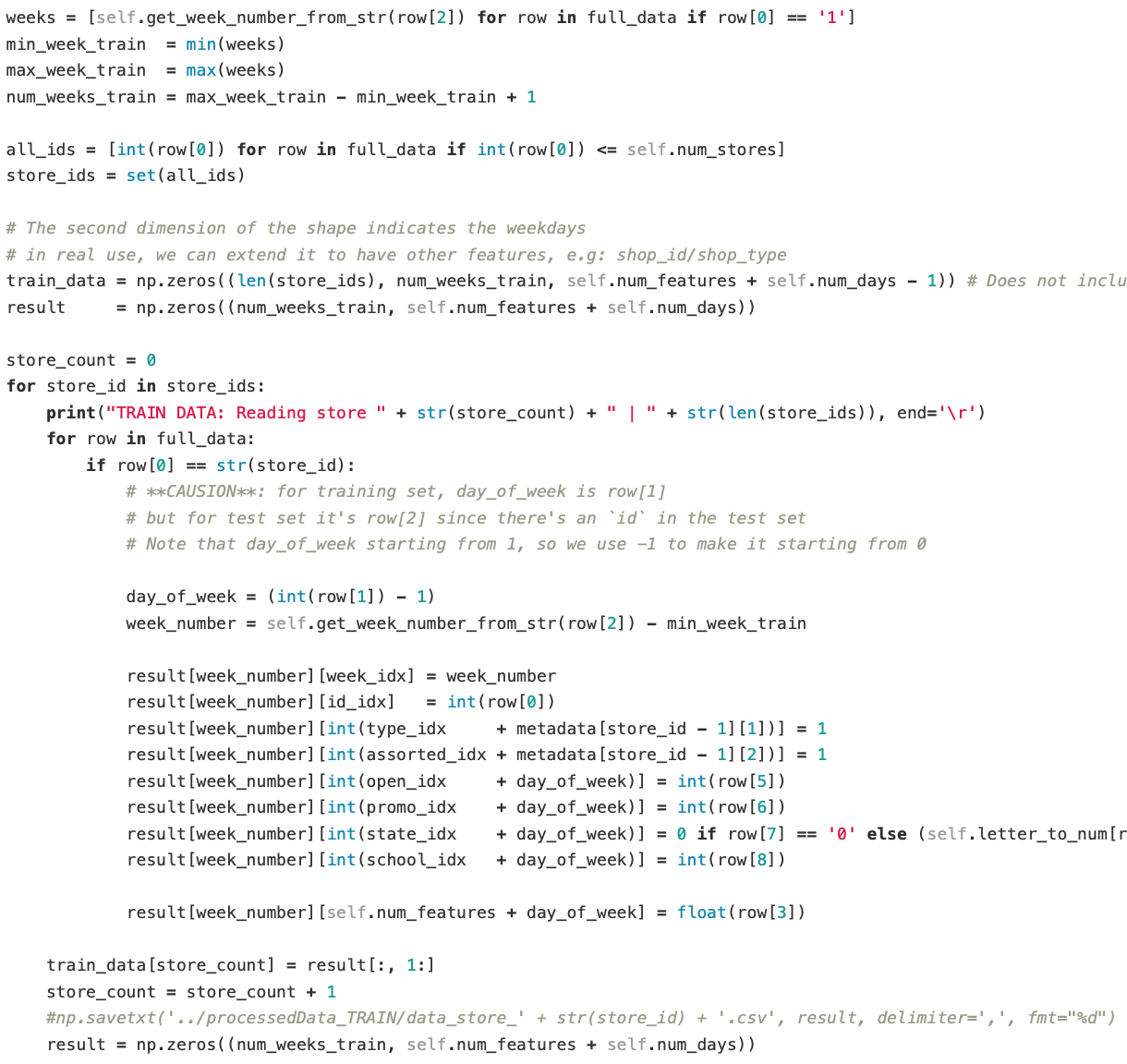




In the first instance the DataFolding class is called from the LSTM class to get training and test data. This takes three csv files for metadata, training and testing respectively as well as the number of stores to operate upon and the training files are store.csv for metadata, train,csv for training and test.csv for testing. The class is then constructed (above) with different directories for the metadata, training and test data. The main purpose of the class is to fold day by day data into weeks of training set data to be sampled by the LSTM class. It also takes care of examples where there is only a day or two of sales data or so called one hot examples.

In the first instance the metadata is processed first by separating each of the comma delimitated for temporary vector to be operated upon when folding the test or training data. The features of the metadata include: week index; store id; store type; whether the store was open or closed; whether a promo was on or not over a week; whether a state holday was on or not; and whether it was a school holiday or not.

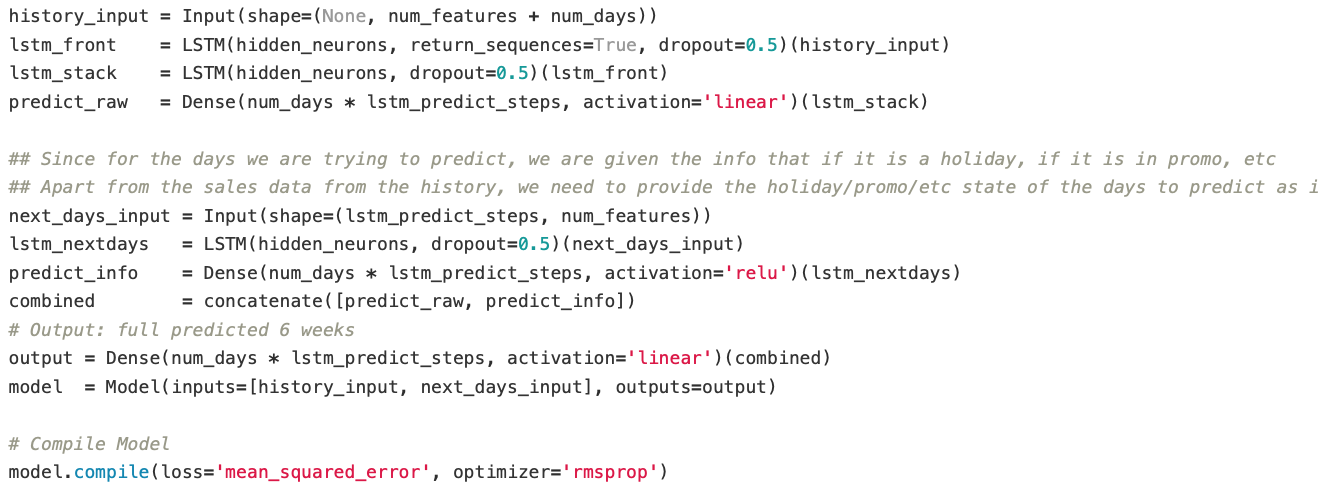




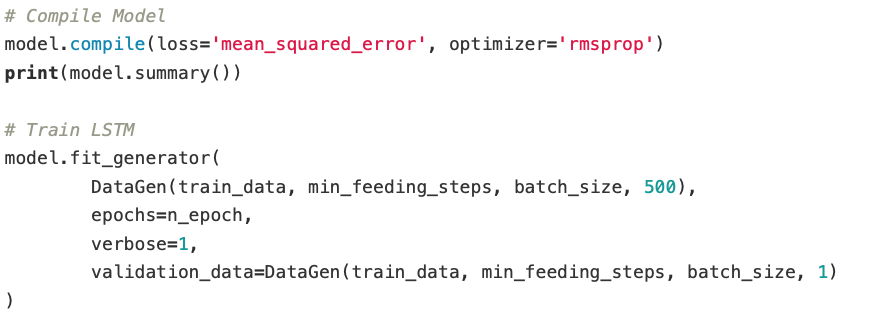
When creating the training or test sets the number of weeks is considered and the max and min number of weeks is calculated based on how many weeks are held in the week vector with the week number being extracted from a date format in a full\_data list of data by day. This gives the amount of weeks to be trained upon for the training set or test set. The final part of generating a training set is to generate a list of store ids that are iterated through to append things like week number, promotions, open days, state and school holidays and other metadata. The generation of a test set follows a similar process to that of creating a training set.

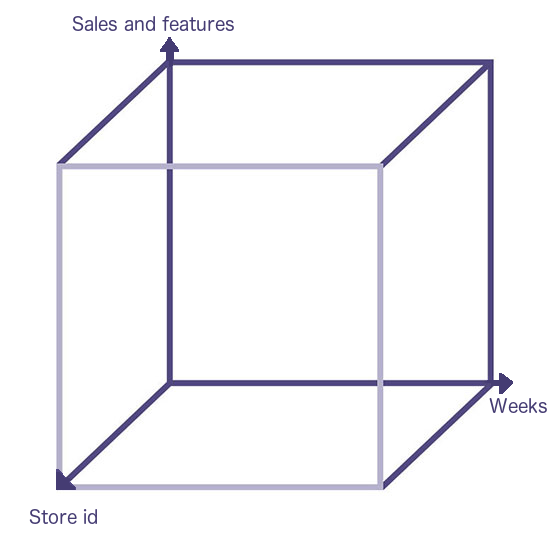
## figure 10: LSTM stacking

In the LSTM file the LSTM cells are stacked together in the format as pictured above. The bottom layer takes in current metadata in (next days input) and the top takes in sales and metadata for past days and weeks (history input).



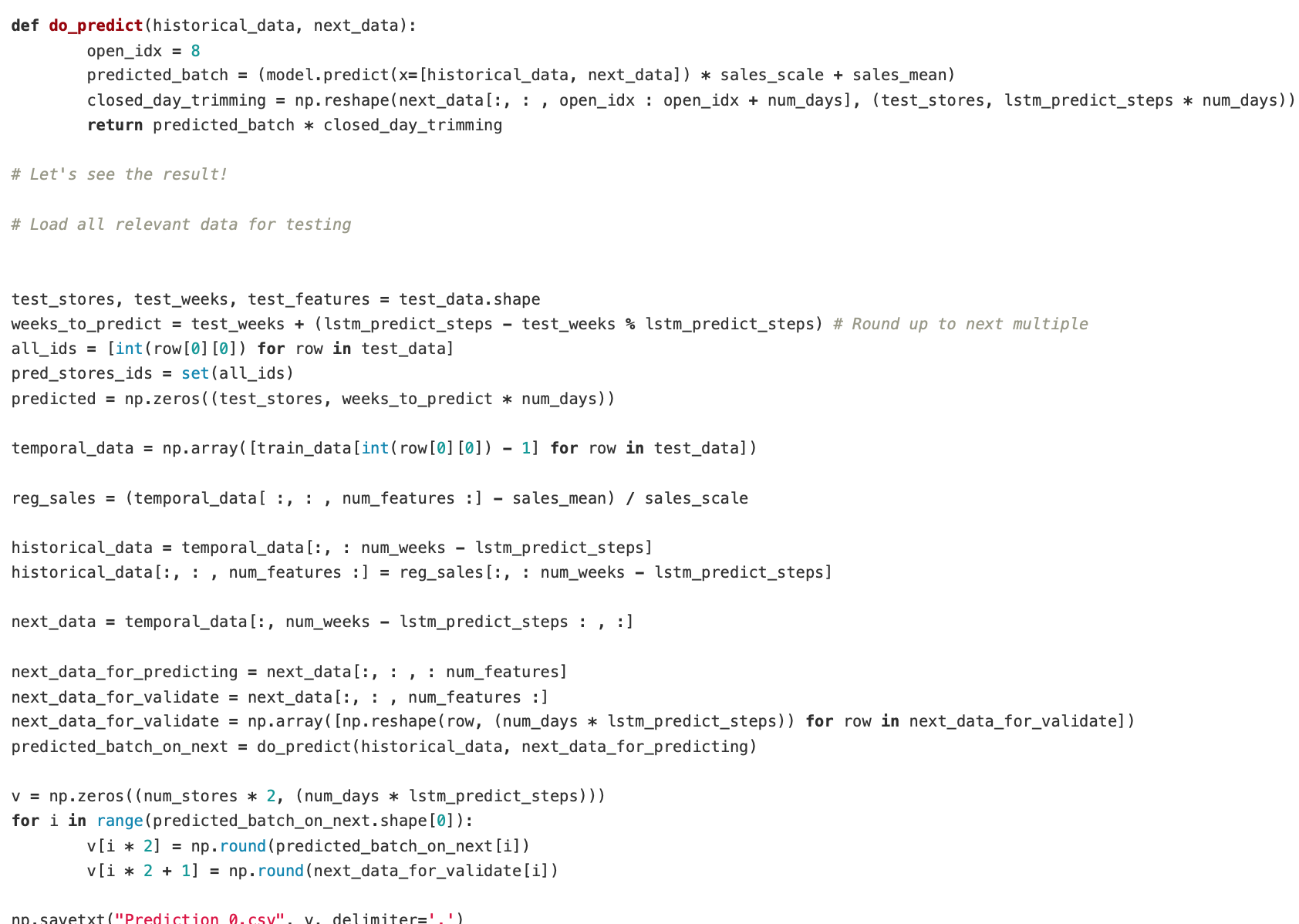
The code (above) demonstrates how inputs are taken for the historical and next days input for each respective branch of the RNN architecture. Following the LSTMs there is a layer of dense neurones that take in LSTM output [14]. The reasoning behind having dense layers is because the LSTMs provide output in TanH. This is a problem because we cannot know ahead of time what the upper and lower bounds future sales and therefore this would create a problem if any sales produced a result that was out of the range -1 to 1 that TanH provides. The two branches are then concatenated together to return a single tensor instead of two tensors [15]. The output dense layer is then applied with a linear activation function to predict the full six weeks (seven truncated into six) before being encapsulated into a model. Then the model is compiled with mean\_squared\_error and rmsprop, which was chosen out of possible alternatives (stochastic gradient descent, momentum, Adam) as it is more suited to time series data with recurrent networks. It is worth noting at this stage that BPTT is done implicitly by Keras.





*fig 11. training set space.*

When the model is fitted to the data it creates a training set or test set space that occupies a three dimensional space of axis (above). For training the DataGen class effectively takes a chunk or a sample out of the total space and each week is processed in batches. The same is true for test data and validation data.



The final lines of code exist to load test data and to do predictions before saving the predictions as a .csv file. The result is a set of predictions and actual values that can be compared together. In the results below it is possible to see the cases where good or poor performance was achieved by the prediction algorithm, with orange representing predicted values for sales data and the blue line representing the actual sales data.



Here it is possible to see that the second store and thirteenth store that there is a rather large over estimation of the number of sales. Whereas the prediction for the fourteenth store is more or less accurate.



Here also the estimations seem to be much tighter or more approximate to the actual sales than that for the second or thirteenth stores. It could be said that from examining these graphs that the overall performance gain was good to excellent. What the data demonstrates is to the layman quite obvious: promotions, school holidays and Saturdays all have a positive impact to sales. It is patterns like these that which may seem obvious, but experiments like the one constructed are needed to verify.

The individual contribution of the author to the Kaggle project was to perform the initial research on key elements of the architecture and approach such as LSTMs versus GRUs as well as LSTM architecture. As mentioned previously this lead to the choice of LSTMs over GRUs because of better documentation for LSTM solutions. In the end a minority of the team members did any if at all coding. The remaining work for everyone else on the project was to research and document all key concepts such as LSTMs, GRUs, BPTT etc.

# Conclusions and future work

The first part of this report covered the literary review for both the Kaggle competition and the neural network self driving robot. The experiment was then covered in both cases by explaining what the code actually does before comparing the test results from both. In both cases there is either very good (Kaggle) or quite good results (robot) with plenty of potential future work for the robot to make it perform better. If you recall what the behaviour was like in the outset for the robot it would just move in circles and not follow any wall on the left hand side. If 50 or more neurones were added it would then move in a straight line but it would yet again not respond to any obstacles along the way.

The second iteration robot could therefore be considered to be a major improvement over the first iteration. The robot from this iteration actually follows the wall and responds to obstacles along the left hand side to the robot. The only commonality between both iterations is that when obstacles are very far away the robot just moves in circles. There is therefore a lot of work that could be done to improve this by modifying the bias, alpha and other hyper parameters. A lot of the reason why it took so long to implement the second iteration is that a complete interface needed to be written to process the input gathered from the left wall following PID robot. It also required writing a lot of helper functions as aforementioned to work out SGDM, normalisation of -1 to 1 and for bias. This lead to overall what could be called quite good performance as a whole, but could be developed further to avoid obstacles more readily by turning faster and to make sure that the robot notices far away objects as a potential left hand wall to follow. However, it is worth noting that the success of the Kaggle competition experiment supersedes even the results of the second iteration of the self driving robot.

##### References

[1] H. Hagras, "Lecture 3: Neural Network Architecture and Learning", University of Essex, 2018.

[2] A. Sharma, "Understanding Activation Functions in Neural Networks", *Medium*, 2017. [Online]. Available: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0. [Accessed: 22- Dec- 2018].

[3] J. Collis, "Glossary of Deep Learning: Bias – Deeper Learning – Medium", *Medium*, 2017. [Online]. Available: https://medium.com/deeper-learning/glossary-of-deep-learning-bias-cf49d9c895e2. [Accessed: 22- Dec- 2018].

[4] H. Hagras, "Lecture 4&5: Feedforeword Neural Networks Using Supervised Learning", University of Essex, 2018.

[5] N. Donges, "Gradient Descent in a Nutshell – Towards Data Science", *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0. [Accessed: 22- Dec- 2018].

[6] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, 1st ed. MIT press, 2016, pp. 296-299.

[7] N. Buduma, *Fundamentals of Deep Learning*, 1st ed. Sebastapol: O'Reilly, 2017, p. 20.

[8]I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. London: CRC Press, 2009, pp. 646-659.

[9]A. Walia, "The Vanishing Gradient Problem – Anish Singh Walia – Medium", *Medium*, 2017. [Online]. Available: https://medium.com/@anishsingh20/the-vanishing-gradient-problem-48ae7f501257. [Accessed: 22- Dec- 2018].

[10]F. Chollet, *Deep learning with Python*, 1st ed. New york: Manning, 2018, p. 196.

[11]J. Chung, C. Gulcehre and K. Cho, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling", NY, 2014.]

[12]T. Shen, "GRUs vs. LSTMs – Paper Club – Medium", *Medium*, 2018. [Online]. Available: https://medium.com/paper-club/grus-vs-lstms-e9d8e2484848. [Accessed: 23- Dec- 2018].

[13]J. Brownlee, "A Gentle Introduction to Backpropagation Through Time", Machine Learning Mastery, 2017. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-backpropagation-time/. [Accessed: 23- Dec- 2018]

[14]"Rossmann Store Sales | Kaggle", *Kaggle.com*, 2015. [Online]. Available: https://www.kaggle.com/c/rossmann-store-sales. [Accessed: 23- Dec- 2018].

[15] “Merge Layers - Keras Documentation", *Keras.io*, 2019. [Online]. Available: https://keras.io/layers/core/. [Accessed: 07- Jan- 2019].

[16] “Core Layers - Keras Documentation", *Keras.io*, 2019. [Online]. Available: https://keras.io/layers/merge/. [Accessed: 07- Jan- 2019].