COSC420 Neural Networks

Nicholas Dong 6729754

Introduction

Life on earth has existed for billions of years. Over time, organisms arose, adapted, and evolved into the living beings that inhabit the planet today. One of the primary drivers for evolution was the need to solve problems. The answer to this was the development of a heavily connected computational structure that we know as the brain. With the use of arrays of sensory receptors, multiple layers of processing units, and complex integration of all their signals, animals with brains are able to make complex decisions in response to any problems they face.

The first neural networks were created in an attempt to imitate the problem-solving potential of the human brain (Kriesel, 2005). They are comprised of processing units named "neurons", named after the processing cells in the animal brain. Neurons are connected by directional weighted connections, that allow propagation of outputs. These structures allow us to solve large and complex problems by breaking them into many small problems, and combining their solutions to solve the initial large problem. Neural networks come in all shapes and sizes and are suited for different functions. For example, the basic neural network design is the feedforward perceptron, which has ordered layers of processing units that receive an input, and produce a logical output. The present report focuses on a multilayer perceptron, which has the capability to solve complex logical problems. Another network design is the fully connected Hopfield network. Rather than processing an input into an output, the Hopfield network has active neurons that influence one another to find a stable network state. Hopfield networks and similar designs can be used for pattern recognition and completion (Kriesel, 2005). Modern designs such as convolutional networks are still being researched and optimized, showing that there are still many things to find out about, in this growing field.

Neural networks are set apart from other computational counterparts by their ability to learn. Because of the way the neurons are connected, the final outputs of the network can be changed by simply altering the connection weights. Weights can be changed according to a desired "teaching input", general reinforcement cues, or even according to internal rules set by the network itself. Given some amount of training, a neural network should be able to calculate a "correct" response or state, much like an animal can choose a correct behavioural response to a stimulus (Nielsen, 2018).

The multi-layered perceptron presented in this report utilises a weight-change paradigm called backpropagation, also known as the generalised delta rule. This is a type of supervised learning, where the desired outputs are known for each set of training inputs. First, training inputs are fed into the input layer of the network, and propagated through hidden layers, until they reach the output layer. This is called the "forward pass". The final outputs of the network are compared to the teaching inputs, and the connection weights from the hidden layers to the output layers are changed according to how different the given output was, in comparison to the desired output. Hidden neurons calculate the change they need to make to their connections by propagating the error of the outputs to their predecessors, during the "backwards pass". After this process is repeated multiple times, the network should be able to correctly calculate the desired output from given inputs (Kriesel, 2005).

The present report will first look at simple tasks such as XOR and parity tasks, which are traditional examples of problems that can be solved by multi-layer, but not single-layer, perceptrons. Different hyperparameters will be investigated, to study the optimal design for these tasks in terms of efficiency and accuracy. Next, the larger Iris data set will be used to study how well the network can generalise new inputs to similar trained inputs, and some methods to improve generalisation.

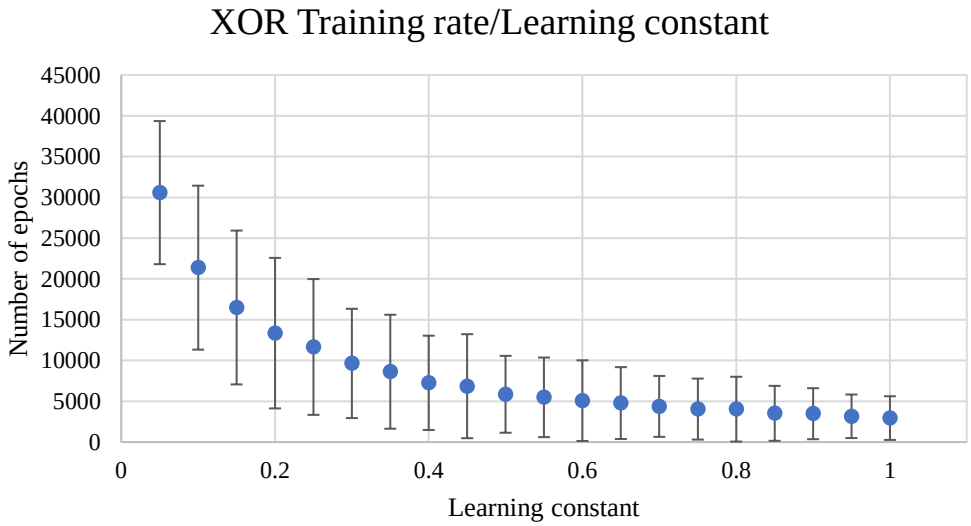
The neurons have a semi-linear sigmoidal activation function, as described in the COSC420 lectures. Sigmoidal activation functions are suitable for function approximation and logical processing problems, such as the ones covered in the present report (Rojas, 1996). Other activation functions include RELU, Maxout, or simple binary activation functions. The error function used is a mean square error calculation, as described in the COSC420 lectures.

XOR tasks

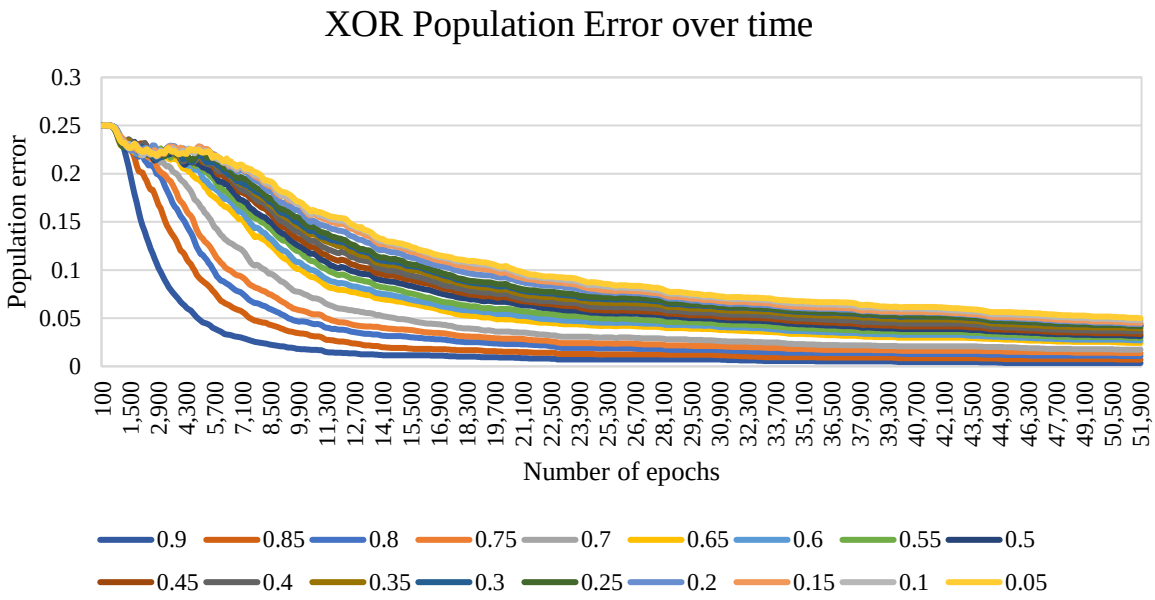
In the XOR task, the network has 2 input neurons, 2 hidden neurons, and 1 output neuron. It is expected that the neuron will output 0 if the inputs are the same, and 1 if the inputs are different. XOR uses binary inputs (0/1). The error criterion used was 0.02 as described in the COSC420 sample parameters. Population error was calculated every 100 epochs using a MSE calculation.

Learning constant

The first hyperparameter to investigate is the learning constant, which is used in the weight change equation to control the rate of learning. Smaller learning constants will likely be slower, but larger learning constants may be less accurate.



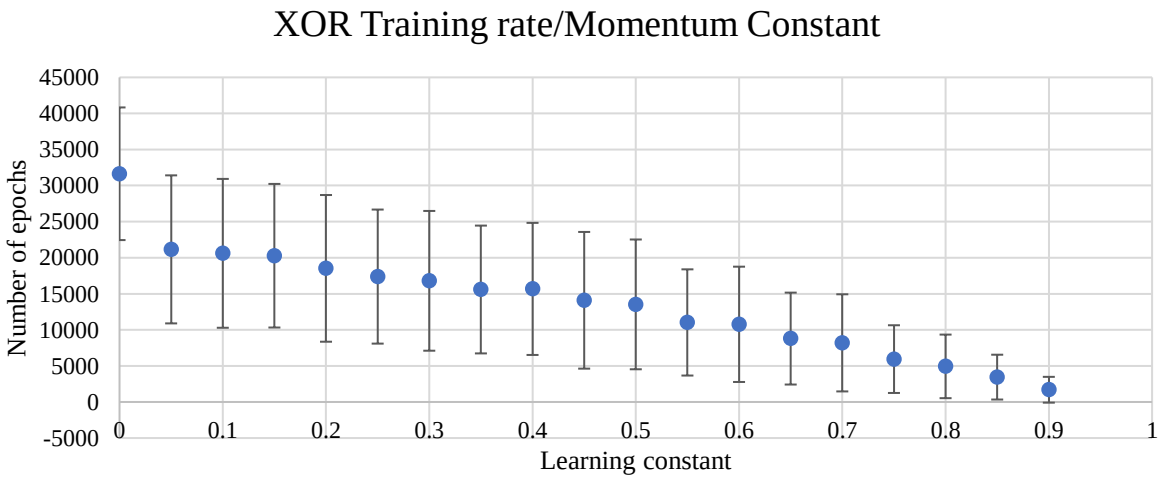
This simulation was run 500 times per learning constant, in a range of 0.05-1.0. As the learning constant increased, the average number of epochs needed decreased, down to an average of 2500 epochs at a learning constant of 1. Higher learning constants result in greater weight changes per epoch, meaning the network can reach the desired set of weights faster, in this case. The downside of increasing the learning constant is a potential decrease in accuracy, discussed later in regard to the Iris dataset.



The line graph above shows the average population error of the network over time, with different learning constants. This visualisation supports that the networks with larger learning constants can reach a lower population error at a faster rate.

Momentum

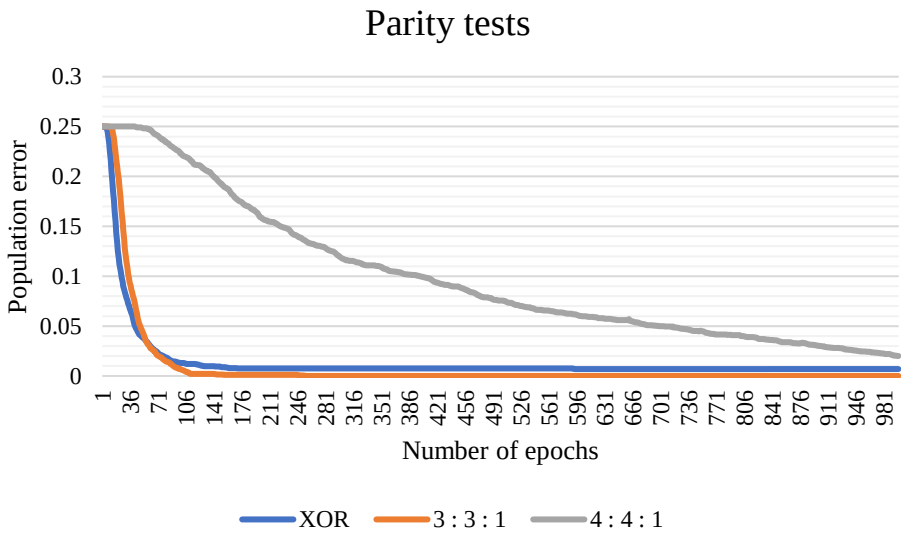
Adding a momentum component into the weight change equation means that some residual change from the last weight change is added to the next weight change. A network with momentum can make successive weight changes in the same direction at a faster rate. This experiment used a range of momentum constants, from 0 to 0.9, and a stable learning constant of 0.1. The error criterion used was 0.02. Simulations were run 500 times.



Even the smallest amount of momentum can improve the rate of learning, seen from the drop in number of epochs needed to learning, from the momentum constant of 0 to 0.05. Higher momentum constants appear to speed up learning further. Note that momentum constants greater than 1 will cause numerical explosion as the network attempts to exponentially increase weight change, which causes an overflow error.

Further parity tasks

XOR is an example of a parity task, where the network wants to determine if a certain number of input neurons are active in a pattern, rather than the arrangement of the inputs. Below are the 3:3:1 and 4:4:1 networks, alongside the 2:2:1 XOR network. Simulations were run 500 times with a learning constant of 0.1, a momentum constant of 0.9, and error criterion of 0.02.



The feedforward neural network is able to solve the 2:2:1 and 3:3:1 parity tasks in less epochs than the 4:4:1 parity task. This is likely due to the increase in information from the number of connections in the 4:4:1 task making it more difficult to learn. The increase in dataset size may require more calculation to find the correct weights for the task. Different hyperparameters may also have different effects on the different datasets.

So far, the population error has been calculated in regard to data that the network has been trained with. Most neural networks are intended to work on data that the network has not been trained on, and categorise unfamiliar information using the trained weights. This is a feature of neural networks called generalisation. The following experiments on a larger data set will allow us to test how well the network generalises.

Iris Tasks

The Iris dataset contains 150 patterns, comprised of features of iris flowers. Based on these features, the neural network should be able to categorise the patterns into three species of irises. The dataset calls for a 4:3:3 network, where three output neurons correspond to the three separate species. The output neuron with the greatest activation tells us which species the flower belongs to. This dataset is large enough for us to train the network on a subset of data rather than all of the data, and then test its performance on previously unseen patterns.

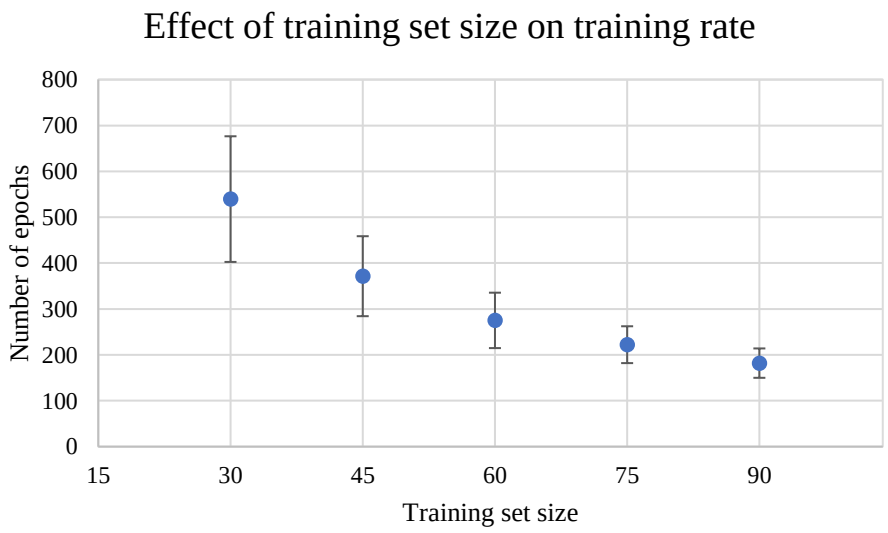
Initial testing

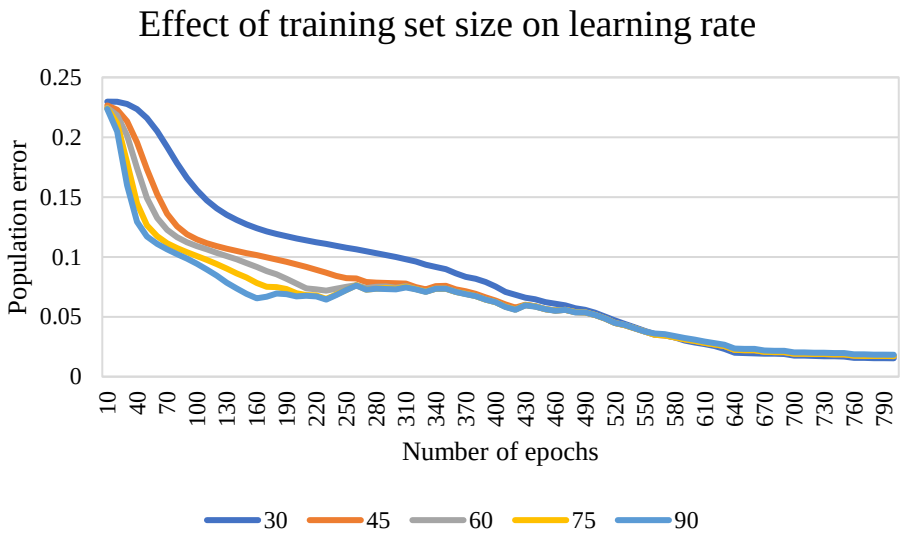
First, by training the network on the entire dataset, it was observed that the network could learn to categorise within an average of 300 epochs, using a learning constant of 0.1 and a momentum constant of 0.9. The network was altered to look at population error every 10 epochs, finding an average of 234 epochs to reach an error criterion of 0.1.

Next, the network was altered to separate the dataset into three subsets: a training set, which it could cycle through during each epoch and adjust weights accordingly; a validation set, to which it could check population error and the end of each epoch; and a test set to check generalisation after learning was completed. I/O pairs were randomly shuffled before the subsets were generated.

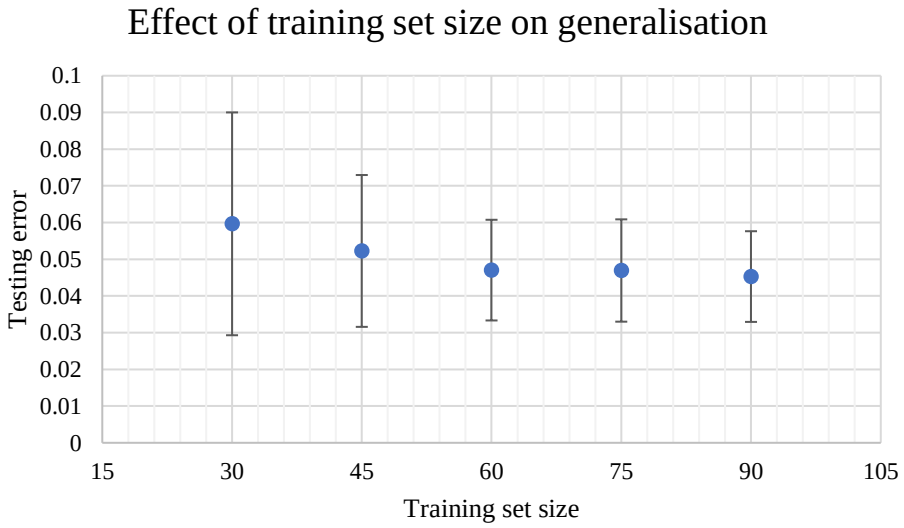
Training set size

The first test looks at the effect of training set size on learning and generalisation. Training, validation, and test sets were set to a 70:15:15 ratio. Training data sets ranged from 30 to 90 data points. The learning constant used was 0.1, with no momentum. The error criterion used was 0.02. Simulations were run 50 times. Population error was calculated from the validation set, while testing error was calculated from the test set after training was completed.





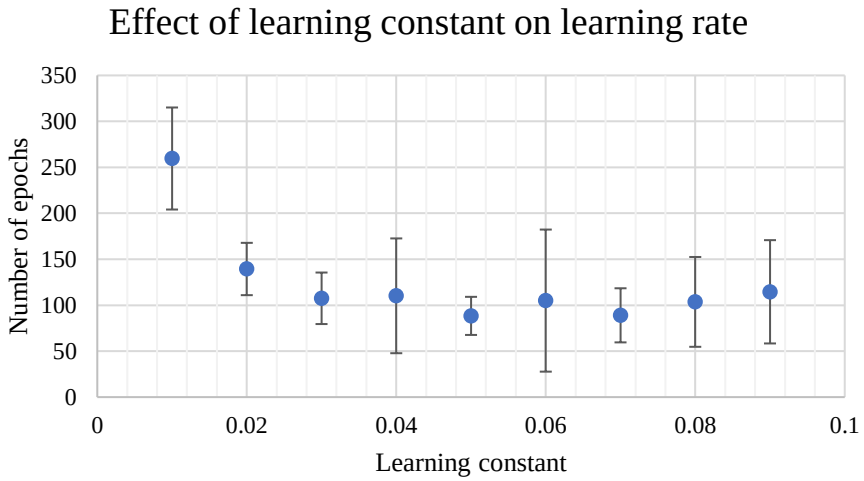
Having a larger training set allows the network to learn faster, as there is more data to compare with and make changes from. This would allow for greater amounts of change to happen in a single epoch. Note that this dataset was able to be solved using few data points. This only works when the data subset is representative of the entire dataset, which may not be possible for larger datasets with more output "categories".



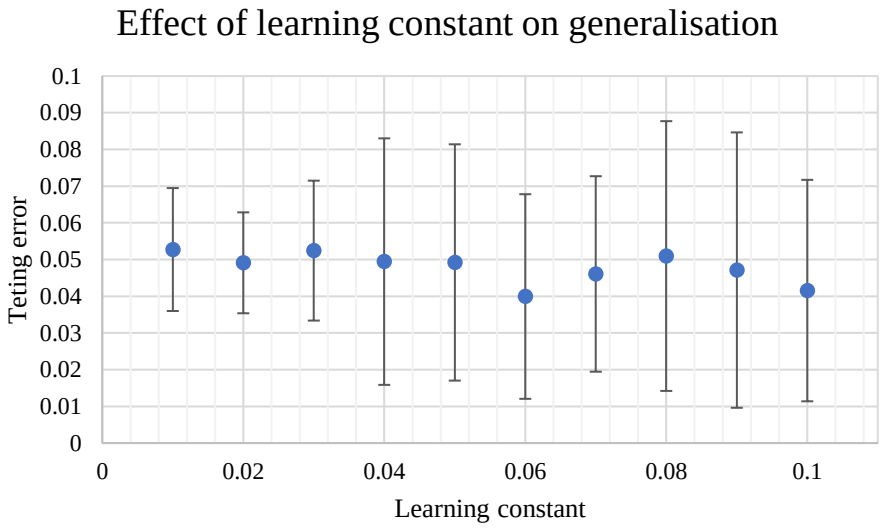
Contrary to expectations, the smaller dataset also had decreased generalisation, as seen in the above graph. Normally it would be expected that the faster training would result in impaired generalisation. This could have been caused by the size or nature of the dataset affecting learning. While the change does not appear to be very large at first, a paired T-test between the testing errors of the 30 and 90 groups found that the difference was significant ($p < 0.05$). In this example, the testing error difference may not be very impactful, but in data sets that require higher specificity and generalisation is vital, larger training sets could be beneficial. Methods to counteract this include cross-validation of a small training set. The network can separate the training set into a training and validation subset, then recombine and separate again with a different selection for the validation subset. Additional permutations of the data act as additional data sets for training, which can then lead to increased generalisation ability.

Learning constant

This experiment used a range of small learning constants to investigate the interaction between learning rate and generalisation ability. It was hypothesised that using a larger learning constant would increase the rate of learning, at the cost of generalisation. The learning constants used ranged between 0.01 and 0.1. No momentum was used. The error criterion used was 0.02. Simulations were run 50 times.



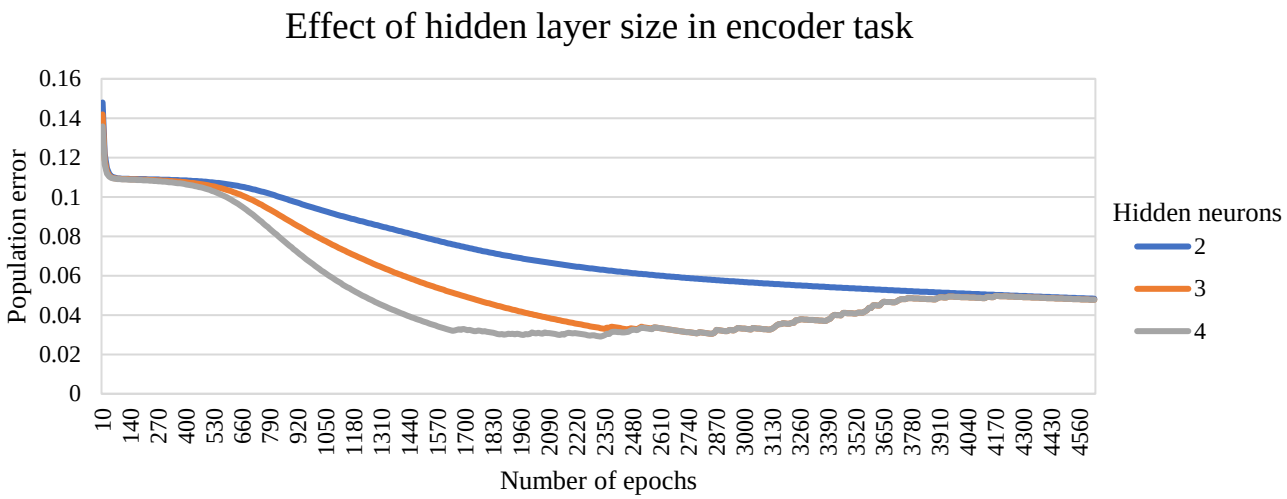
The number of epochs required for the testing error to drop below 0.02 decreased from 250 to roughly 100, as the learning rate increased. However, the network took over 2000 epochs to reach a testing error of 0.02 with a learning constant of 0.1, showing that it could not quite generalise to the same extent.



While not immediately apparent, a t-test showed that the testing error of the 0.01 and 0.1 networks was significantly different ($p < 0.05$). The higher learning constant network had slightly lower generalisation ability. This can be due to the sharper weight changes creating "steeper gradients on the error surface" (Kriesel, 2005) and the network is less able to fit unfamiliar data to the trained data.

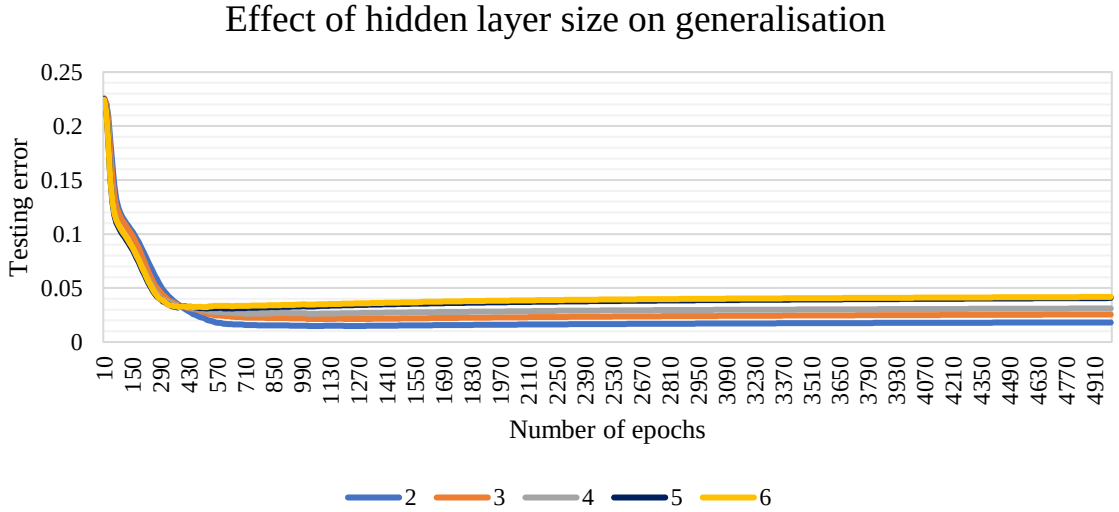
Network size

The network must have the specified number of input and output neurons to match the dataset and desired output. The number of neurons in the hidden layer, or layers, can vary. To first show the difference in training rate, an encoder protocol was tested. In the encoder test, the teaching input is the same as the input – the hidden neurons must be able to associate input of one input neuron to its corresponding output neuron. 8 inputs were mapped onto 8 input neurons and passed to 2, 3, or 4 hidden neurons, which calculated outputs for 8 output neurons. A learning constant of 0.1 was used, with no momentum. Simulations were run 50 times.



The network learns faster with more hidden neurons, with the population error reaching 0.5 first with 4 hidden neurons. The network decreased population error to as low as 0.3, with 4 hidden neurons. After training for a while, the population error appears to rise again and joins with the population errors from the networks with 2 or 3 hidden neurons. This may be due to some unexpected properties of the encoder task, or a form of overfitting.

The next experiment used the Iris dataset, and a range of 2-6 hidden neurons. To show the progression of learning, 60 data points were used, with a learning constant of 0.1 and no momentum. Simulations were run 50 times.



The networks with more hidden neurons learned in the shortest time. However, after an extended period of training, their testing error begins to increase, showing a decrease in generalisation ability. This is due to a phenomenon called “overfitting”, where the network begins to memorise the training set data rather than calculating categories. Because of this, it has an impaired ability to categorise the unfamiliar testing set data. Networks with less hidden neurons are less prone to overfitting, and can retain generalisation ability. Other methods to avoid overfitting and keep generalisation include stopping training early, and adding noise to obscure the training data. This is also why a validation set is used to calculate population error for larger datasets, rather than using the pre-learned training set data.

Discussion

Neural networks are a powerful tool for problem solving in computing. The studies in this report begin to introduce just a few of the customization options for the multi-layer perceptron, one of the basic neural network designs. It should be obvious that programmers need to consider the design of the network, the data set, and the intended purpose of the network, during the creation of the network.

The learning constant is often considered to be the most important hyperparameter (Kriesel, 2005). The experiments on the smaller networks showed that larger learning constants will increase the rate of learning. Experiments on the larger Iris dataset show that larger learning constants have the risk of overlearning the training data set, resulting in lower generalisation. Furthermore, if the learning constant is too large, the network may overcorrect for the connection weight errors and get caught oscillating over and under the desired weight. Networks that require high accuracy may use additional features to speed up the learning process, such as momentum and other extensions.

Momentum is a tool for increasing the rate of learning without leading to oscillation (Swingler, 1996). It is used during the weight change calculation to amplify changes in the same direction as the previous change, or weaken changes in the opposite direction. The experiments in this report show that momentum can make a noticeable difference in increasing the efficiency of learning, and is a relatively simple extension to implement.

Various guidelines have been published for recommendation of the size of neural networks, and the number of hidden neurons to be used. Too few, and the network will fail to converge. Too many, and the network will overfit the training set and be unable to generalise. Generalisation is the ability of a neural network to solve problems containing unfamiliar data. It can be tracked during learning with the use of a testing subset, which is a known I/O pair excluded from the training subset. Networks are often designed with generalisation in mind, if a programmer wants to design a versatile product with wide-ranged application.

The amount of data used in training is another point for consideration. If the training set is small, the network may not be able to extrapolate enough assumptions to unfamiliar data, and will not be able to generalise. On the other hand, the network will be able to learn the small data set very well, and will be highly accurate for assumptions within the given data.

There seems to always be a trade-off for each aspect of neural network design. This report outlined a multi-layered perceptron and some methods to balance its efficiency and generalisation. Clients must be specific about their desired end-product, in order for programmers to create the perfect neural network design for their needs.

Appendix

The neural network consists of 4 java files: NeuronLayer, Neuron, Connection, and NeuronApp. When NeuronApp is opened in the compiler, it reads the given files (param.txt, in.txt, and teach.txt) to set the hyperparameters and fill in arrays to be used as the input, and teaching input.

Using the hyperparameters, the program will begin building a neural network. First, three NeuronLayers are initialised, with empty ArrayLists to hold the neurons. Next, Neurons are initialised and placed into these NeuronLayers, according to the amounts specified in param.txt. Additionally, a bias neuron with a constant output of 1 is initialised.

Connections between Neurons are created, linking the network together. These Connections are initialised with random weights with a value of ±0.3. Connections are added to each Neuron's ArrayLists of input connections and output connections, so the Neurons can keep track of which other Neurons they are connected to. Connections from the bias neuron are initialised with a weight of 0.

The following text will appear in the console:

> Neural network constructed. Enter one of the following commands:
>
> (L)earn - learn weights according to given teaching pattern.
>
> (T)est - test population of input patterns, and see activation of all units.
>
> (W)eights - show weights of connections between neurons.
>
> (E)xit - exit program.

Entering either the command or the first letter of a command will select that option.

When (L)earn is selected, the network takes the input array and sets the input neuron activations to the inputs specified in the first pattern. Hidden neuron and output neuron activation is calculated. Each output neuron's output is compared to the corresponding teaching input to get an error term, which is applied to its incoming connections to calculate the necessary weight change for the connection via the delta rule. This error term is backpropagated to the hidden neuron layer, where each neuron summates the error terms of each output neuron, multiplied by the weight of the connection. This allows it to calculate the weight changes for its own incoming connections. Weight change occurs offline at the end of each epoch, so the calculated changes are simply stored by the connections for now. This process is repeated for every pattern in the input array. Once the end of the epoch is reached, connection weights are adjusted. Population error is calculated every 100 epochs, and is printed, along with the number of epochs. If population error is less than the desired error criterion, the network stops learning. Otherwise, the process continues to repeat, until reaching error criterion or 500,000 epochs.

When (T)est is selected, the network will run through the input patterns and show the network's outputs for each pattern, showing the activation of all neurons.

When (W)eights is selected, the network will display the list of current connection weights, including which neuron they come from and which neuron they connect to.

When (E)xit is selected, the program terminates.