University of Otago

Department of Computer Science

COSC420 Assignment Report

COSC420 Assignment 1: Neural Networks

Anthony Dickson (3348967)
May 3, 2019

Abstract

Neural networks are a family of complex learning algorithms. One facet of this complexity is choosing the correct hyperparameters such as learning rate, momentum, and batch size. For this assignment I perform a large scale parameter search and explore the effects of certain hyperparameters and other various settings. Through my experiments I investigate the validity of several rules of thumb regarding these hyperparameters and settings, and the majority of them prove to be sound advice in the context of the Iris data set.

1 Introduction

Neural networks have proven to be very capable learners [1, 2, 3] but choosing the correct topology and combination of hyperparameters still remains more of an art than anything else. There are no universally applicable rules to derive the optimal configuration of a neural network, and most rules of thumb seem to be unreliable at best. This leads to single approach - trial and error. While one can build a good intuition with experience to what kind of neural network would be necessary for a particular problem, increasingly complex problems become difficult to reason about. To further make things worse, neural networks have become increasingly complex and computationally demanding as high-performant neural networks have become deeper and deeper with millions of parameters (think GoogLeNet [4]), making it costly to train and evaluate said neural networks.

Fortunately for this assignment we are only looking at two layer multi-layer perceptron networks trained on small data sets so we do not have to worry about this too much. This means a large number of neural network configurations can be trained, evaluated, and compared. For this assignment I train 4,536 unique neural network configurations with 40 trials each, leading to a total of 181,400 individual neural networks. These were all trained and evaluated over 800 CPU hours on a 12-core machine (which is close to three days in real time or close to a month on a single-core machine). The number of unique configurations is the result of performing a grid search over various values for eight hyperparameters and other various settings. Due to the sheer amount of data and the complexity of said data, in this report I will only explore a subset of this data.

The rest of this document is structured as follows: first I cover what settings I explore, then I detail my experiment setup, then I explore the effects of these settings, then I conclude my findings and give a final discussion about my assignment work. Additionally there are two appendices that detail how to run my code and list some supplementary data.

2 Explored Hyperparameters & Other Settings

In this assignment I explore the effects of eight different hyperparameters and settings. These are: the data set and neural network topology; the amount of noise added to the inputs; the batch size, or how the data samples are presented to the network; whether or not the data is shuffled at the beginning of each training epoch; the representation of the learning task; the activation function used in the hidden layer; the learning rate; and momentum. I will discuss each of these individually and detail the range of values

that I explore in my experiments.

2.1 Data Sets & Network topologies

I train neural networks on the six data sets that were provided in the sample data for this assignment and use the indicated neural network topologies. Some example patterns are shown in Figure 1. I will denote the neural network topologies with three numbers representing the number of input layer neurons, the number of hidden layer neurons, and the number of output layer neurons. The data sets and the neural network topologies are as follows:

- 3-bit parity: the data set for which a 3:3:1 network must learn to correctly choose the parity bit for a 3-bit pattern
- 4-bit parity: the data set for which a 4:4:1 network must learn to correctly choose the parity bit for a 4-bit pattern
- 535: the simple data set with two five-dimensional patterns for which a 5:3:5 network must learn to simply associate the two pattern sets.
- encoder: a data set for which a 8:3:8 network must learn to first compress and then recover a 8-bit pattern
- iris: Fischer's Iris flower data set for which a 4:3:3 network must learn the classification of three different species of Iris flowers
- xor: the exclusive OR data set for which a 2:2:1 network must learn the XOR logical operator.

I chose to explore all six data sets because I want to see if there are rules of thumb that I can derive from my results, and also compare the effects of certain parameters across different network topologies and data sets.

2.2 Adding Noise to Inputs

I experiment with perturbing the inputs with what is called additive white Gaussian noise during training. This is simply adding small amounts of Gaussian distributed noise with zero mean and finite variance such that the overall distribution and shape of the data is unchanged. It is said that adding noise to the inputs can improve generalisation [5] and it also can be seen as a form of regularisation [6] or data augmentation [7]. I experiment with three different standard deviation values:

- 0.00 - This is the control value and it is equivalent to not adding any noise.
- 0.01 - This is the small amount of noise. Since values in the data sets range from zero to one this should have a small effect on the inputs.
- 0.10 - This is the large amount of noise. Considering the range of values in the data sets, this should have a large effect on the inputs.

2.3 Batch Size

Neural networks are generally trained with stochastic gradient descent (SGD) which has three main variants based on how the population of samples are presented during a single epoch of training. I explore online SGD where the samples are presented one-by-one, batch SGD where all of the samples are presented at once, and mini-batch SGD - the middle ground between batch and online learning - where samples are

1 1 1 0 0 → 1 1 0 0 1	1 1 → 0	1 1 1 → 1
(a) 535	(b) xor	(c) 3-bit parity
1 1 1 0 → 0	0 0 0 0 0 0 0 1 → 0 0 0 0 0 0 0 1	0.224 0.624 0.067 0.043 → 1 0 0
(d) 4-bit parity	(e) encoder	(f) Iris

Figure 1: Example input-output patterns from each of the data sets used in this assignment.

presented in batches that are any size between 1 and N, where N is the population size. In this assignment I experiment with batch sizes of 1, 2, and N for all data sets, and for the Iris data set I also experiment with batch sizes of 8, 16, and 32.

I implement these variants all as mini-batch SGD, since online and batch SGD can be thought of as special cases of mini-batch SGD. This means that a mini-batch size of one is equivalent to online SGD and a mini-batch size of N is equivalent to batch SGD. There are also two methods of aggregating the accumulated gradients in mini-batch SGD: summing the gradients and taking the average gradient. In my implementation I take the average gradient for each batch. I choose to explore batch size because it is said that: online SGD is good at escaping local minima; batch SGD is computationally faster and supposedly provides gradients that are more robust to noise and allow for higher learning rate constants; and mini-batch SGD supposedly combines the advantages of both of these methods [8, 9, 10]. The authors of [10] also say that small batch sizes of 32, or even small as two or four, perform the best. So I will investigate whether or not these claims hold.

2.4 Shuffling the Data

Another recommendation is to shuffle the data at the start of each epoch. This affects the order in which samples are presented and the data distribution of batches in mini-batch SGD, however it should be noted that this has no effect on batch SGD. I experiment with both shuffling and not shuffling the data at the beginning of each training epoch. One idea that I explore is that shuffling the data each epoch when using mini-batch SGD helps prevent getting stuck with bad batches that would hinder learning and thus improves the overall performance of a neural network.

2.5 Learning Task Representation

Under the supervised learning paradigm there are two main types of learning tasks: regression and classification. The main difference is that in regression we try to predict continuous values whereas in classification we try to predict discrete values. This also affects the structure of neural networks used for these learning tasks. I experiment with both regression and classification models which necessitates two different network structures.

For regression tasks neural networks use the identity activation function in the output layer, loss (or cost/error) is measured using root mean

squared error and *goodness* is measured with Pearson's correlation coefficient.

For classification there are two types of models depending on the number of classes that are to be predicted. In the case of two classes neural networks have a single output neuron that uses the sigmoid activation function, loss is measured with binary cross entropy, and goodness is measured with accuracy. In the case of three or more classes there are K output neurons that use the identity activation function, the output of the layer is transformed with the softmax activation function, loss is measured with categorical cross entropy and goodness is measured with accuracy.

I have setup the classification models that use the softmax activation so that they predict a single label. This has the consequence that the 535 data set cannot be learned by these classification models, since it requires multiple labels to be predicted for a single pattern. As such the results of classification models trained on the 535 data set are invalid and will not be used.

It will be interesting to see how learning task representation affects the neural networks' performance. In general I would expect the classification type models to perform better than the regression models, since they effectively shrink the output space. However, it will be difficult to directly compare the two types of models due to the different measures used for evaluating performance.

2.6 Hidden Layer Activation Function

The activation function used in the hidden layers is also another decision one must make when designing a neural network architecture. I experiment with two of the most popular activation functions, the rectified linear unit (ReLU) and the sigmoid activation function.

For the networks that use the ReLU activation function I use the Leaky ReLU variant. With this variant negative activation values are multiplied by a scaling factor rather than being assigned to zero:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

where x is the activation value and α is the scaling factor. This prevents potential issues that arise to the hard zero gradient of the ReLU activation [11, 12], which is often dubbed the 'dying ReLU problem'. In my experiments I set α to 0.1.

It will be interesting to see how the ReLU activation function's unbounded output affects learning as opposed to the effects of saturation in the sigmoid activation function.

2.7 Learning Rate

Learning rate affects how much of a gradient a neuron will use to update its weights. Choosing a bad learning rate (i.e. too low or too high) can adversely affect learning, and even prevent the network from learning, and as such is an important hyperparameter to get right. There are methods for adaptive learning rates where the learning rate is changed over time according to some criteria. I, however, will look at constant learning rates and experiment with learning rate constants of 0.001, 0.01, and 0.1.

2.8 Momentum

Momentum is a nifty trick to expedite SGD. The basic momentum method that I use adds a proportion of a given neuron's previous weight update to its current gradient. The weight update

with momentum can expressed as such:

$$\Delta \mathbf{w}_{ji}(t) = -\eta \frac{\partial E}{\partial \mathbf{w}_{ji}(t)} + \alpha \Delta \mathbf{w}_{ji}(t-1)$$

where $\mathbf{w}_{ji}(t)$ is the value of the weight connecting unit j to unit i at the time step t, η is the learning rate constant, $\frac{\partial E}{\partial \mathbf{w}_{ji}(t)}$ is the partial derivative of the error function with respect to the weights, $\Delta \mathbf{w}_{ji}(t-1)$ is the amount the weights changed at the previous time step and α is a momentum constant in the interval $[0.0, 1.0)$.

One rule of thumb is to use a momentum constant of 0.9 [7]. In my experiments I test momentum constants of 0.0 (i.e. no momentum), 0.5, and 0.9 to evaluate this claim.

2.9 Section Summary

In this section I detailed the different parameters and settings that I evaluate in my experiments. In summary I evaluate neural networks trained with: six different data sets; three different levels of added noise; three different batch sizes and six different batch sizes in the case of the Iris data set; shuffling and not shuffling the data at each epoch; classification and regression representations of the learning task; two different activation functions in the hidden layer; three different learning rate constants; and three different amounts of momentum. This gives a total of eight sets of parameters and 4,536 unique combinations of these settings.

3 Experiment Setup

To evaluate the different configurations (combination of hyperparameters and settings) I employ a grid search like evaluation scheme. More formally, it generates a list of unique configurations by taking the Cartesian product of each of the parameter sets:

$$C = S_1 \times \cdots \times S_n$$

where C is the set of unique configurations, and S_i is a set of parameters (e.g. $S_6 = \{\text{ReLU}, \text{Sigmoid}\}$ in the case of the activation function settings). An example configuration would be: Iris data set, Gaussian noise $\sigma = 0.1$, batch size of 1, shuffle data at the beginning of each epoch, regression learning task representation, ReLU activation, learning rate of 0.1, momentum constant of 0.9.

The grid search algorithm steps through each of these unique configurations and evaluates each of them over 40 independent trials. In the case of the Iris data set I use 8 times repeated 5-fold cross validation which makes up the 40 trials.

I train each configuration for a maximum of 10,000 epochs and I use two types of early stopping. The first type stops training if the population score is above a given criterion. The second type stops training if the loss does not improve (decrease by at least 1e-5) over the course of the previous 100 epochs. This was done to cut down the training time and since I trained over 180,000 individual neural networks this saved a lot of time. However, it does affect the results which I will discuss in the next section.

For each configuration I record four metrics: the training loss, training score, validation loss, and validation score for each epoch. The validation metrics are only recorded for the Iris data set since the other data sets are not large enough and are not expected to generalise for unseen data. I also record how many epochs the configuration was trained for and details about the configuration itself. This data is then processed and analysed.

There is one other implementation detail that I should also discuss which is the weight initiali-

sation scheme. I first set the weights to random Gaussian distributed values with zero mean and unit variance. I then apply a heuristic called Xavier initialisation [13] where I multiply the random weights by the square root of the number of inputs to the neuron. That is to say:

$$\mathbf{w}_{ji} \sim N(0, \sqrt{\frac{1}{n}})$$

where $\mathbf{w}_{ji}$ is the weight connecting the unit j to the unit i, and n is the number of inputs to the unit i. This is meant to help prevent gradients from vanishing or exploding. This usage is also slightly different to the cited authors' original implementation since they use uniformly distributed weights in the interval $[-1, 1]$ instead.

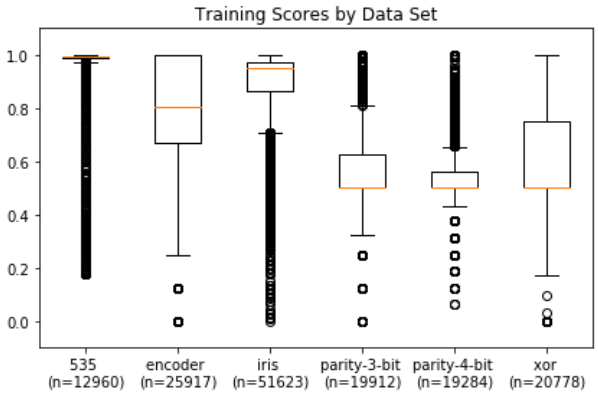


Figure 2: Box plot of training scores by data set. The whiskers extend to a maximum of 1.5 times the inter-quartile range and the points outside this range are plotted as circles and represent outliers. Median values are listed in Table 1 of Appendix B.

4 Results & Discussion

I generate experiment data for all of the unique configurations that were previously detailed, however due to the sheer size and complexity of analysing and drawing comparisons between all the data, I will focus on the results for the neural networks trained on the Iris data set.

There is also one more thing I must mention before going into the results. As mentioned before the regression type models use Pearson's correlation coefficient (PCC), which outputs a value in the interval $[-1, 1]$. This makes it difficult to directly compare the scores from regression models with the scores from classification models, whose score is measured as an accuracy measurement in the interval $[0, 1]$. For the purpose of comparing the scores of these two types of models, I will normalise the scores of the regression models to the range [0, 1] as well. This still does not solve the problem completely since a PCC score of -1 does not mean the same thing as an accuracy score of zero, but I believe it will suffice.

In the rest of this section I will present and discuss the results pertaining to each hyperparameter/setting that I evaluated in the context of the Iris data set, and evaluate the validity of any heuristics that I mentioned in Section 2. Then at the end I will summarise these findings.

4.1 Data Sets & Network topologies

The models are able to do well on half of the data sets but do rather poorly on the other half. The data sets that the models do well on are the 535, encoder and Iris data sets, whereas the data sets that the models do poorly on are the 3-bit parity, 4-bit parity and the XOR data sets. This is shown in Figure 2. It seems that my implementation struggles with the data sets where the output is a single binary value. In fact about 50-60% of the models trained on these three data

sets get stuck at 0.5 training score. This strikes me as odd and may be a sign that something is wrong in my implementation of the neural networks. However, that would then not explain why the models were able to learn the Iris data set rather well. This issue is prevalent across all configurations so I think it is unlikely that hyperparameter settings are to blame. One possible explanation is that models are more likely to get stuck on these data sets, and that my method of early stopping simply terminated training early because of this.

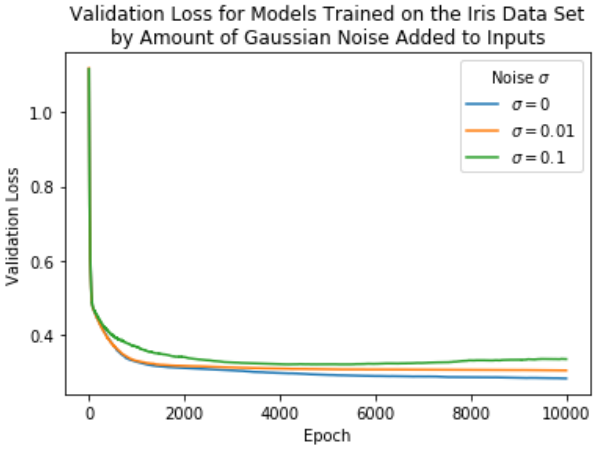


Figure 3: Plot of median validation loss by amount of Gaussian noise added to the inputs.

4.2 Adding Noise to Inputs

Adding noise to the inputs did not show any substantial positive effect on validation scores for models trained on the Iris data set (see Table 2 in Appendix B). In fact, adding noise had a small negative effect on the validation scores.

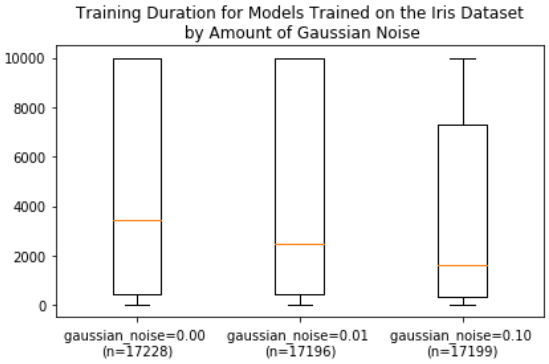
An interesting thing to note is that higher levels of noise are correlated with a shorter training duration (see Figure 4a). This appears to be due to the models with the highest level of noise (standard deviation of 0.1) beginning to overfit and their validation loss starting to increase (see Figure 3). So in the case of the Iris data set and the amounts of noise that I tested, it seems that adding noise does not improve generalisation as claimed in [5].

4.3 Batch Size

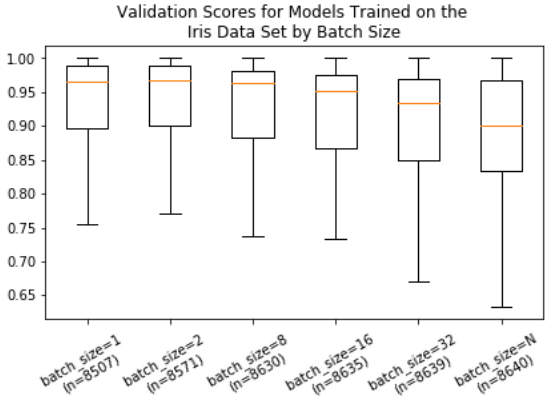
Batch sizes of 1, 2, and 8 perform the best, closely followed by the batch size of 16. The sizes of 32 and N do not perform as well, as show in Figure 4b. It would seem like this data agrees with the statement that smaller batch sizes give better model performance.

Just looking at the results where the models are trained with a learning rate of 0.1 actually shows the opposite. The batch sizes of 8, 16, 32 and N do the best, with batch sizes of 1 and 2 performing slightly worse as shown in Figure 4c. This seems to support the idea that larger batch sizes can support and require higher learning rates. Considering these results and the ones above, it would seem that mini-batch SGD does indeed take the best aspects from both online SGD and batch SGD [8, 9, 10].

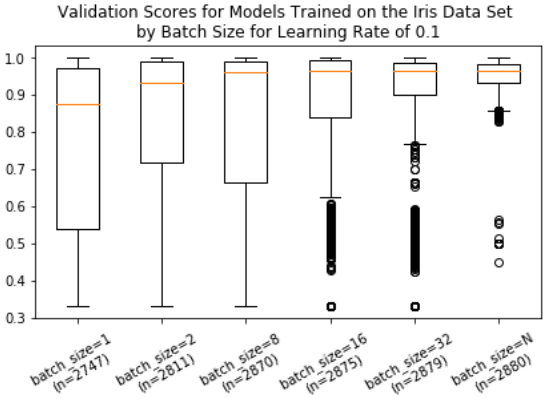
A smaller batch size also turns out to be correlated with a shorter training duration in epochs (see Figure 4d). There is a substantial difference in the training duration between the batch sizes of 1 & 2 and the other batch sizes, with batch SGD needing the full 10,000 epochs in most of the cases. This might provide an explanation as to why a higher batch size is correlated with a lower validation score, as a learning rate higher than 0.1 may be needed in order for the models with higher batch sizes to converge within the set limit of 10,000 epochs.



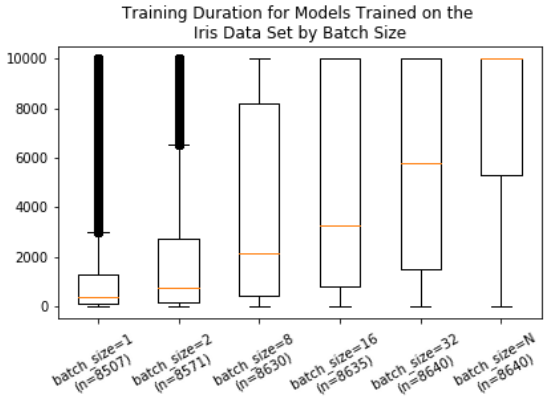
(a) Training duration by Gaussian noise. Median values are listed in Table 3 of Appendix B.



(b) Validation scores by batch size. Median values are listed in Table 4 of Appendix B.



(c) Validation scores by batch size with a learning rate of 0.1.



(d) Training duration by batch size. Median values are listed in Table 5 of Appendix B.

Figure 4: Various box plots of models trained on the Iris data by amount of Gaussian noise added to the inputs (Figure 4a) and by batch size (Figures 4b, 4c, and 4d). Outliers and have been omitted to make the differences between batch sizes more clear in Figure 4b.

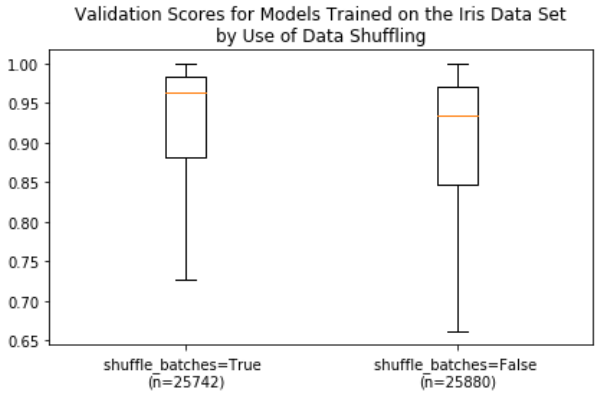


Figure 5: Plot of median validation loss by whether or not the data was shuffled at the beginning of each epoch. Median values are listed in Table 6 of Appendix B

4.4 Shuffling the Data

Shuffling the data at the beginning of each epoch showed an increase in median validation scores of about 3% for shuffling data versus not shuffling data, as shown in Figure 5. This supports the idea that shuffling the data at the beginning of each epoch improves the performance of mini-batch SGD, possibly by avoiding getting stuck with bad batches. There was no substantial evidence suggesting that shuffling the data shortens training.

4.5 Learning Task Representation

The classification models outperform the regression models by about 4.4% in validation scores and also train in about half the number of epochs (see Figure 6a and Figure 6b). I think that this is the logical outcome since the classification models have a simpler task to learn since the output space is significantly smaller due to the outputs being discrete values, as opposed to being continuous values as in the case of regression tasks. Additionally, classification is a much more natural representation of the data sets since they all deal with discrete output. So the experimental setup is rather biased towards the classification representation.

4.6 Hidden Layer Activation Function

The sigmoid activation shows a 1.8% increase in median validation score (Figure 6c) however the median model using sigmoid in the hidden layer takes close to four times longer to train (Figure 6d). This is likely due to the the sigmoid activation imparting a smaller gradient than the ReLU activation.

4.7 Learning Rate

There were no surprising results in regards to the learning rate. A learning rate of 0.1 gave marginally better performance over a learning rate of 0.01, and a learning rate of 0.001 was slow to converge and had a slightly worse median validation score (Figure 7a). The fastest learning rate to train was 0.1, and each subsequent learning rate took approximately ten times longer to train (Figure 7b), although I am only guessing that the models trained with a learning rate of 0.001 would take that long since training stops well before that stage. As mentioned in Section 4.3, with larger batches sizes you can afford higher learning rates. It seems that a good rule of thumb for the learning rate would be to start high and gradually decrease the learning rate if it does not work out – in fact this is exactly the idea behind adaptive learning rates. So I think it would be sensible to use adaptive learning rates where possible.

4.8 Momentum

Using a momentum constant of 0.9 gives the shortest training out of the three settings (Figure 7d) while achieving almost exactly the same median validation score as the other two momentum constants of 0.0 and 0.5 (Figure 7c). This is interesting because it means that there are no apparent downsides to using momentum, at least in this case. The data also agrees with the claim that 0.9 is good rule of thumb for setting the momentum constant [7], although it would have been good if I had experimented with other values such 0.8 and 0.99.

4.9 Notes on the Pearson Correlation Coefficient

It turns out that the Pearson Correlation Coefficient (PCC) has a few issues when applied to the evaluation of the regression models that I explored. The first issue is to do with online SGD where we look at one sample at a time. In this case you cannot calculate the PCC to evaluate the goodness of a single prediction by the network. This is due to terms cancelling out which should be clear if we take a look at the equation for calculating PCC:

$$r_{ty} = \frac{\sum_{i=1}^{n}(t_i - \bar{t})(y_i - \bar{y_i})}{\sqrt{\sum_{i=1}^{n}(t_i - \bar{t})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where t denotes the teacher outputs, y the output predicted by the network, and $\bar{t}$ and $\bar{y}$ are the averages of these. If we are only looking at a single sample then it follows that $t_i = \bar{t}$ and thus $t_i - \bar{t} = 0$, likewise for y and $\bar{y}$. This leads to zeros everywhere and thus the PCC is undefined due to zero division, even if the network had predicted $y \approx t$.

A similar issue occurs when all of the labels are the same for either of the teacher or predicted labels. For example, in the XOR problem if we are using a batch size of two and get a batch of the inputs [0, 1] and [1, 0] then we would expect the outputs to be 1 and 1, i.e. $t = \{1, 1\}$. Evidently $\bar{t} = 1$ and thus $t - \bar{t} = 0$ and once again we get zeros and an undefined PCC score.

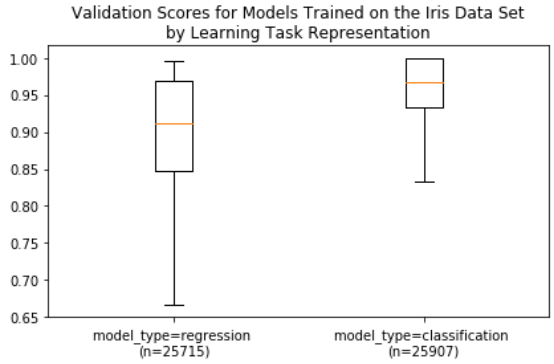
Unfortunately I did not realise this until after I had run all of my experiments. Nevertheless, a single question remains: is there a better measure of goodness for regression tasks, similar to accuracy used for classification tasks? It would seem that there are no measures that give a score in the interval [0, 1] like accuracy for regression models. The next best metric I could find is one based on the negative squared euclidean distance which gives a score of one for values that match exactly and a possibly infinitely large negative number for predictions that very far away from the teacher output. This is defined as:

$$f(t, y) = 1 - \sum_{i=1}^{n}(t_i - y_i)^2$$

where n denotes the number of dimensions in the output pattern.

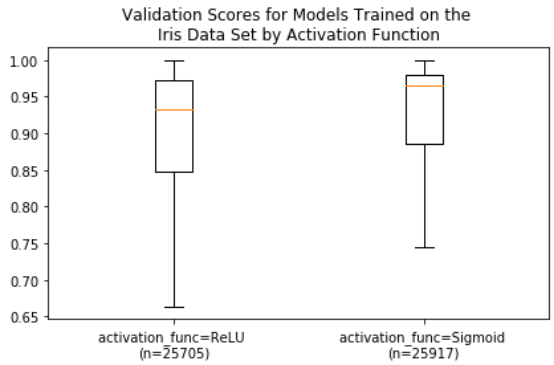
4.10 Summary of Results

In this section I mainly discussed the results for the models trained on the Iris data set. The configurations I experimented with were largely successful at learning the 535, encoder, and iris data sets, however they struggled with the 3-bit parity, 4-bit parity and xor data sets. Using the Pearson correlation coefficient for a measure of goodness for regression models does not work in practice, and an alternative measure based on the negative squared euclidean distance is suggested.

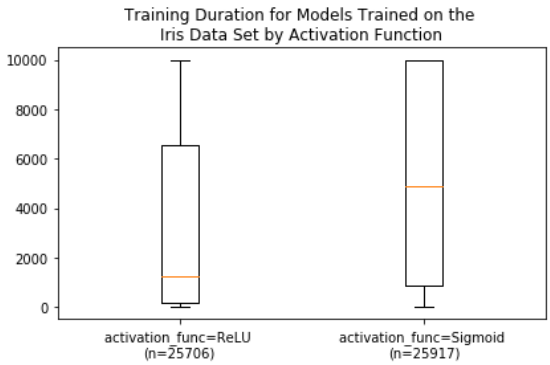


(a) Validation scores by learning task representation. Median values are listed in Table 7 of Appendix B.

(b) Training duration by learning task representation. Median values are listed in Table 8 of Appendix B.

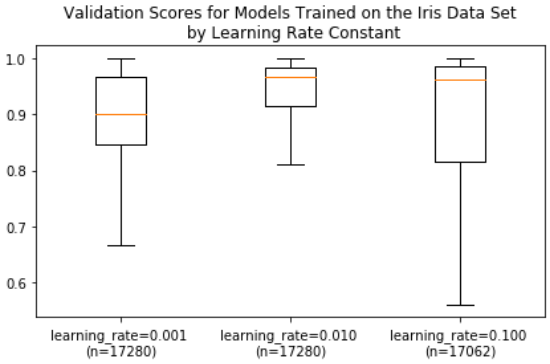


(c) Validation scores by activation function. values are listed in Table 9 of Appendix B.

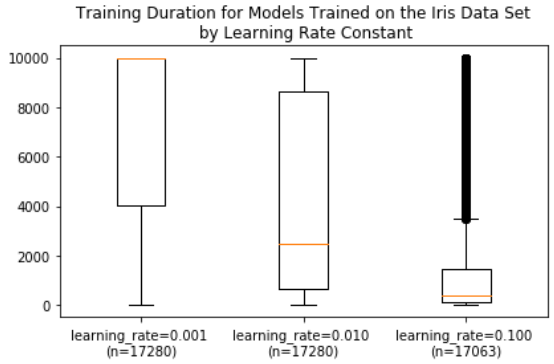


(d) Training duration by activation function. Median values are listed in Table 10 of Appendix B.

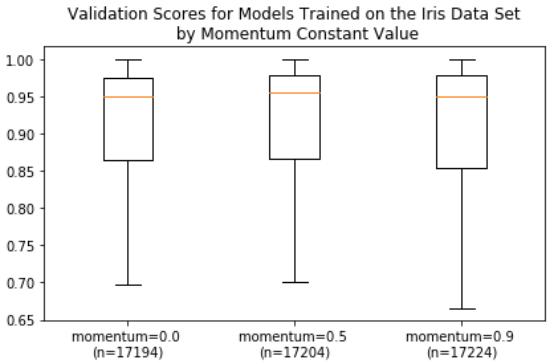
Figure 6: Various box plots of models trained on the Iris data by learning task representation (Figures 6a and 6b) and by activation function (Figures 6c and 6d). Outliers and have been omitted to make the differences between batch sizes more clear in Figure 6a and 6c



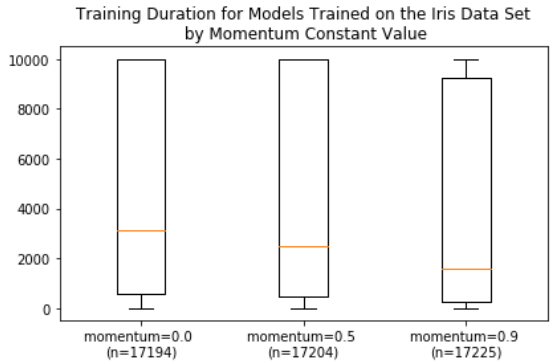
(a) Validation scores by learning rate. Median values are listed in Table 11 of Appendix B.



(b) Training duration by learning rate. Median values are listed in Table 12 of Appendix B.



(c) Validation scores by momentum constant. Median values are listed in Table 13 of Appendix B.



(d) Training duration by momentum constant. Median values are listed in Table 14 of Appendix B.

Figure 7: Various box plots of models trained on the Iris data by learning rate (Figures 7a and 7b) and by momentum constant (Figures 7c and 7d).

Most of the heuristics that I set out to evaluate have proven to be sound advice, namely using small batches, higher learning rates with larger batches, shuffling the data at each epoch, and using a momentum constant of 0.9. However, it is doubtful whether or not adding noise to the inputs actually helps. The choice of activation function at the hidden layers does not seem to be all that of an important decision in these small neural networks.

Overall, the ranking of importance of the hyperparemeters and settings that I tested seems to be: learning rate followed by momentum, learning task representation, shuffling the data each epoch, batch size, activation function, and finally adding noise to the inputs.

One thing that I thought would be interesting is to use my results from the Iris data set to try come up with some sort of best configuration. Going off my results I came up with the following configuration:

- a learning rate of 0.1
- a momentum constant of 0.9
- using a classification type model
- a batch size of 2
- the sigmoid activation function
- shuffling data at each epoch
- and no noise added to the inputs.

The proposed model is compared against all of the other configurations in Figure 8 and it appears to be much better than the other configurations, with a median training score of 0.9333 compared to the median training score of 0.8175 of the other configurations. However, these numbers should taken with a grain of salt since I am comparing about 200 samples against 150,000 samples.

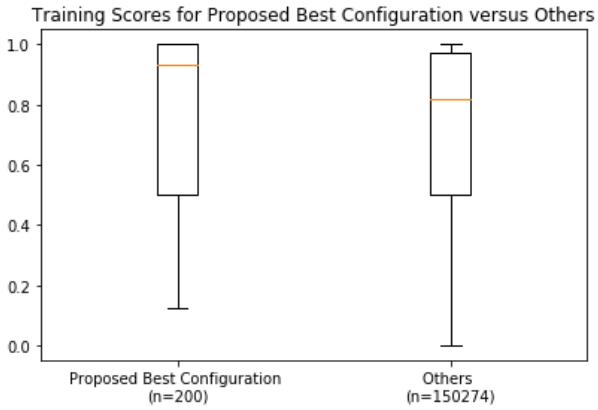


Figure 8: Box plot of training scores of the proposed best configuration compared against all other configurations. Median values are listed in Table 15 of Appendix B.

5 Future Work

Future work would include addressing the issues brought up in the previous sections and exploring the rest of data. Things that I would change for future experiments would include things such as: changing the target outputs to 0.05 and 0.95 for models using the sigmoid activation in the output layer; using the binary cross-entropy loss function for multi-label outputs such as the 535 data set instead of the categorical cross entropy loss; running my experiments again without early stopping; and experimenting with smaller amounts of Gaussian noise to perturb the inputs with.

The size of the results data was many times the size of the memory available on my machine, so it made handling the data cumbersome espe-

cially since I was trying to work with a monolithic csv formatted file. This made it very difficult to produce time series plots of model performance over the training epochs since the uncompressed data was around 55GB. If I were to run the experiments again I would put all of this data into a relational database. This would also reduce the overall size of the results data because a large proportion of the data is array padding since numpy does not seem to allow jagged arrays. This evident since the 55GB of raw data compresses down to about 9GB. Overall, saving the results data into a database should make exploring the data a lot more manageable and reduce the space needed to store them substantially.

Another thing that could be interesting would be to save the weights of all of the trained models. This is manageable since the weights are typically less than 1KB for these small models. This way one would be able to load the models that, for example, struggled on the xor data set and examine the aspects such as the activation values to see why they got stuck at 0.5 training score so often.

6 Conclusion

For this assignment I performed a large scale parameter search to collect data about the performance of various hyperparameter and design choices regarding neural networks. I explored the results for models trained on the Iris data set and raised a few issues regarding how my experiments were set up and carried out. I also evaluated several heuristics regarding the settings I tested and found the majority of them to be accurate.

References

[1] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[2] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *URL https://openai. com/blog/better-language-models*, 2019.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[5] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.

[6] Chris M Bishop. Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[8] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[9] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[10] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[11] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[12] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

A Running the Code

In this section I will detail the dependencies necessary for running my code, the files in my submission, and how to run the demo code.

A.1 Dependencies

My neural networks are implemented in python. I use a few packages such as numpy, scikit-learn, and matplotlib. The python packages needed to run my code are listed in the included `environment.yml` file. If you are using a distribution of conda you can install the packages with the command:

```
$ conda env create -f environment.yml
```

which will create a conda environment called `cosc420`. You can change the name field in the `environment.yml` file before running this command if it conflicts with a preexisting environment. Otherwise, you may need to go through and install the packages manually.

The program in `backprop_demo_graphviz.py` uses Graphviz to visualise the forward and backward passes of a neural network, so if you want to run this demo program then you will also need to install Graphviz. However, it is not required for text-based demo program in `backprop_demo.py`.

A.2 File Structure

My submission has the following folder structure:

```
/
├── data
│   ├── 535
│   │   ├── in.txt
│   │   ├── teach.txt
│   │   └── params.txt
│   ├── ...
│   └── xor
│       ├── in.txt
│       ├── teach.txt
│       └── params.txt
└── demos
    ├── backprop_demo.py
    ├── backprop_demo_graphviz.py
    ├── iris_classification.py
    ├── iris_model.json
    ├── xor_classification_model.json
    └── xor_model.json
```

```
    │   ├── xor_regression.py
    │   └── mlp
    │       ├── activation_functions.py
    │       ├── datasets.py
    │       ├── layers.py
    │       ├── losses.py
    │       └── network.py
    ├── environment.yml
    └── README.md
```

All of the data sets are located in the `data` folder which contains the data sets:

- 535
- 3-bit-parity
- 4-bit-parity
- encoder
- iris
- xor

and this will be important if you are running the demo code and want to try out different data sets.

The `mlp` folder contains all of the source code that implements the neural networks. Hopefully the names of the files are self-explanatory but I will go over what each of these files contains. In `network.py` I implement the high level functions of a neural network such as the main training loop, providing an interface for predictions, early stopping, splitting data and serialisation. Forward and back-propagation are implemented in the `layers.py` file in the `DenseLayer` class, in the same file the layer that adds Gaussian noise to its inputs is also implemented. The activation functions and loss functions are located in the `activation_function.py` and `losses.py` files respectively.

The `demos` folder contains contains a set of demo applications using the source code from `mlp`. The files `demos/xor_regression.py` and `demos/iris_classification.py` contain demos that train a predefined neural network on a given data set for a maximum of 10,000 epochs, print training and validation statistics every 100 epochs, and finally plot these training statistics. These can be run with no additional arguments or anything but are not interactive.

The program in `demos/backprop_demo.py` is an interactive text-based demo that can be run with most network structures and settings (except for early stopping, the stopping criterion, and batch size). In this program you can specify a neural network model by creating a JSON file (there are three examples made available which give examples of the different settings possible and the expected structure). Then you can run the program and specify which JSON file to

use and which data set to load. Then you can: view the weights; train the network; plot loss history; test input patterns; and step through an epoch viewing the activations of the forward pass, the weight updates in the backward pass, and epoch summary statistics. There is also `demos/backprop_demo_graphviz.py` which has a subset of these features but uses Graphviz to visualise the forward and backward passes of the network as an annotated graph structure.

These demo programs must be run from within demos folder due to some package import hacks I had to use (modifying the path variable used for package imports). The demos `demos/backprop_demo.py` and `demos/backprop_demo_graphviz.py` take in some command line arguments. You can display a help text which explains these by running the scripts and using the help flag, for example:

```
$ python backprop_demo.py --help
```

will display a help message explaining the expected arguments. By default these demos will run the XOR data set with the model defined in `demos/xor_model.json`. Essentially for the last two demos you need to: define a neural network model in a JSON file; start the demo program specifying which model and data set to use; follow the on screen prompts which will tell you what commands you can execute.

B Supplementary Tables

Data Set	Median Training Score
535	0.9900
encoder	0.6250
iris	0.9432
3-bit parity	0.5000
4-bit parity	0.5000
xor	0.5000

Table 1: Median training scores of all models by data set.

Gaussian Noise σ	Median Validation Scores
0.00	0.9423
0.01	0.9398
0.10	0.9333

Table 2: Median validation scores of models trained on the Iris data set by amount of Gaussian noise added to the inputs.

Gaussian Noise σ	Median Training Duration
0.00	3488
0.01	2438
0.10	1610

Table 3: Median training duration for models trained on the Iris data set by the amount of Gaussian noise added to the inputs. Lower is better.

Batch Size	Median Validation Scores
1	0.9667
2	0.9667
8	0.9667
16	0.9667
32	0.9428
N	0.8920

Table 4: Median validation scores for models trained on the Iris data set by batch size.

Batch Size	Median Training Duration
1	417
2	758
8	2448
16	3866
32	7815
N	10000

Table 5: Median training duration for models trained on the Iris data set by batch size. Lower is better.

Shuffles Data?	Median Validation Score
Yes	0.9632
No	0.9333

Table 6: Median validation scores for models trained on the Iris data set by use of data shuffling.

Learning Task Representation	Median Validation Score
Regression	0.9221
Classification	0.9667

Table 7: Median validation scores for models trained on the Iris data set by learning task representation.

Learning Task Representation	Median Training Duration
Regression	3858
Classification	1811

Table 8: Median training duration for models trained on the Iris data set by learning task representation. Lower is better.

Activation Function	Median Validation Score
ReLU	0.9487
Sigmoid	0.9667

Table 9: Median validation scores for models trained on the Iris data set by activation function.

Activation Function	Median Training Duration
ReLU	1484
Sigmoid	5867

Table 10: Median training duration for models trained on the Iris data set by activation function. Lower is better.

Learning Rate	Median Validation Score
0.001	0.9333
0.01	0.9667
0.1	0.9667

Table 11: Median validation scores for models trained on the Iris data set by learning rate.

Learning Rate	Median Training Duration
0.001	10000
0.01	2847
0.1	362

Table 12: Median training duration for models trained on the Iris data set by learning rate. Lower is better.

Momentum Constant	Median Validation Score
0.0	0.9665
0.5	0.9667
0.9	0.9667

Table 13: Median validation scores for models trained on the Iris data set by momentum constant.

Momentum Constant	Median Training Duration
0.0	3518
0.5	2848
0.9	1824

Table 14: Median training duration for models trained on the Iris data set by momentum constant. Lower is better.

Configuration	Median Training Scores
Proposed Best	0.9333
Others	0.8175

Table 15: Median training scores of the proposed best configuration and other configurations.