COMP36212 EX3
Gradient Based Optimisation

Paul Vautravers
10461739
The University of Manchester

May 12, 2023

Abstract

SGD was found to be a strong optimisation method for ANN training, with a batch size of 10 and learning rate of 0.1 providing the best balance of accuracy and efficiency. Furthermore, learning rate decay was implemented, with learning rate values decreasing between 0.1 and 0.001 giving accuracies up to 98.42%. It was not possible to find an implementation of momentum to recreate this accuracy, though a combination of $\beta = 0.9$, $\eta_0 = 0.1$, $\eta_N = 0.001$ and batch size 100 yielded results up to 98.48%. Finally, RMSprop was found to be very sensitive to hyper parameter selection and did not improve results from using learning rate decay and momentum.

1 Introduction

Whilst the most formal and abstract problems pose significant challenges to humans, these are often resolved with ease by computers. Conversely, computers struggle enormously with tasks considered trivial for humans. Such a problem may present itself as differentiating between different animal species or between handwritten letters in an ensemble of images. It is natural to wonder then whether computers could be used to solve these human-intuitive problems, near-automatically and on an enormous scale. Artificial intelligence (AI) presents itself as the answer to this question. Artificial intelligence, however, is a general term and could even entail direct coding of knowledge of the world to try and solve these problems. But, generally, AI systems appear to work best when they are able to learn from data, accumulating their own knowledge of the world – performing what is known as machine learning (ML) [1]. The effectiveness of ML systems is strongly contingent on the data representation at hand. These systems can be helped by providing the most relevant features to learn from, but this is not always possible. Sometimes representation learning may be used to identify the most pertinent features of a problem; again, however, representation learning is not infallible, particularly when extracting abstract features from raw data. As Goodfellow et al explain, this problem is resolved by Deep Learning (DL).

The key feature of DL is the formation of representations that build upon other, simpler representations. In this manner, DL methods manage to understand complex heirarchies of features, learning the highest level features by understanding the lower level features which comprise them [2]. Deep learning models are often based off of Artificial Neural networks (ANNs), composed of sequences of layers, formed of nodes, also known as neurons. Deriving in concept from models of cognition, these neurons are connected to neurons of other layers, to differing extents, through 'synapses'. Essentially, this is a mapping of inputs to outputs, using 'hidden layers' with different functions to represent the input in a new manner. In this sense, the additional depth of multiple layers allows the ANN to learn the representation of the data. Here, a multilayer perceptron (MLP) was used, a form of ANN where neurons in a layer are connected to all the neurons of immediately previous and following layers of the network.

As stated by Goodfellow, Bengio et al, most DL algorithms can be broken down into multiple simple elements. There is a dataset (perhaps presented amenably to a computer), there is a cost function that evaluates the error of the algorithm, there is a model whose parameters are learned to produce the correct output, and there is an optimiser that minimises the algorithm error, bringing the model to the closest representation of reality [1]. In this case, an MLP of 5 layers in total will be used, 2 constituting the input and output layers and 3 the hidden layers of the network, with the number of neurons illustrated in figure 1. The MLP will be tested on its ability to correctly classify images of digits present in the MNIST dataset, and the error associated with this classification will be minimised in multiple epochs of training. The MNIST dataset has 70,000 sample greyscale images, each of 28x28

pixels, split into 10000 testing images and the remainder for training. This project will explore how an alternative to classical gradient descent, stochastic gradient descent, can be implemented and subsequently improved upon.

2 Theory

2.1 Artificial Neural Network Architecture

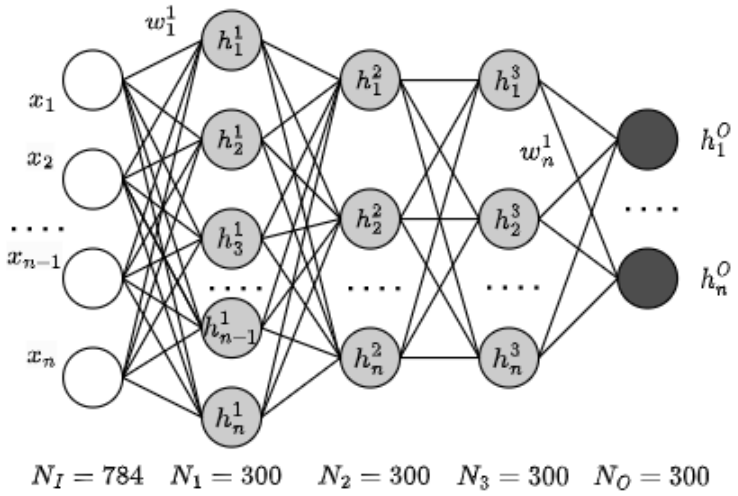


Figure 1: Schematic of the MLP used here, with weights w, inputs x and activations h adapted from the project brief provided by Dr Rhodes [3]. The layers I and O denote input and output layers and N the number of neurons.

The 28x28 pixel input is processed as a flattened array of 784 values, with each pixel intensity value then normalised to the unit interval. Each of these nodes then receive a single pixel intensity, specifying the activation for this layer. The subsequent layers' activations are calculated according to activation functions of each neuron. The activation of neurons in the hidden layers are denoted h_j^n, where n represents the layer index and j the neuron index in the layer. The functions governing activation are different for the hidden layers and output layer, with ReLUs and Softmax used respectively. The activations calculated from ReLU and softmax are presented in equations 1 and 2:

$$h_j^n(h^{n-1}) = \begin{cases} z_j^n, & \text{if } z_j^n \geq 0 \\ 0, & \text{otherwise} \end{cases} \quad \text{where} \quad z_j^n = \sum_{i=0}^{N} w_{ij} h_i^{i-1} \tag{1}$$

As can be seen from equation 1, the activation of a hidden layer n, requires the weighted sum of all activations from the previous layer n-1 to be passed into the ReLU. However, the ReLU does not produce bounded outputs, inconvenient for interpretation of the NN output. As such, the final output layer outputs positive values which sum to 1 across the whole output layer. With a layer width of 10, these outputs then denote probabilities that the sample image belongs to a specific digit. This is achieved by applying the softmax function to the regular activation from ReLU at the output nodes.

$$S_j = \frac{e^{h_j^O}}{\sum_{k=0}^{N} e^{h_k^O}}. \tag{2}$$

As such, the network has produced an estimation of the digit which can be compared against the true data, given the MNIST dataset is a labelled dataset [3].

2.2 Neural Network Loss and Gradient Descent Optimisation

The comparison of the output probabilities with the true label allows a loss function to be evaluated. The label providing the true value of the sample is converted into a vector,$\hat{y_k}$, of length equal to the output layer,N, with only zeros apart from a single 1 value, at the element corresponding to the correct digit. Then, the loss, L, between p_k and $\hat{y_k}$ can be computed using cross entropy loss:

$$L = -\sum_{k=0}^{N} \hat{y_k} log(p_k), \tag{3}$$

Where the summation is over the elements of the two vectors $\hat{y_k}$ and p_k. Note, this loss value is only for a single input. The loss function of the total network requires division of the sum of all samples' loss contributions by the number of samples. To improve the classification performance of the neural network, it is necessary to optimise the weights used in the network, minimising $f(w)$. This requires adjustment of the current weights of the network by a step proportional but opposite in direction to the gradient of the loss:

$$\nabla f(w) = \frac{1}{n} \nabla \sum_{i=1}^{n} L_i(x_i, w) = \frac{1}{n} \sum_{i=1}^{n} \nabla L_i(x_i, w), \tag{4}$$

where L_i denotes the loss term for each sample, indexed by i. The optimisation step then has update rule:

$$w = w - \eta \nabla f(w), \tag{5}$$

where it should be noted that w is a single weight, η is the learning rate hyperparameter and that this equation is applied to all weights for every node [3]. For classical gradient descent, this sum is over all data samples, which can be very computationally expensive. In the case of batch stochastic gradient descent (SGD), the sum over all samples is replaced with a sum over just a subset of the training set, m. Weight updates are now applied after processing each batch, applying updates that average weight updates across different inputs, occuring far more frequently than classical gradient descent. We require the inputs to be randomly arranged so that minibatches provide an accurate representation of the whole sample set.

3 Part 1.1: Implementation of Stochastic Gradient Descent

Batch SGD was implemented through updating the code provided for this project. The file main.c calls hyperparameter and file path arguments to be entered by the user, these are then used in the function `run_optimisation()`, from the file `optimiser.c`. Prior to running the optimisation algorithm on the neural network, the weights in the network are initialised randomly. This is performed by `initialise_nn()`, which sets the weights according to the suggested initialisation from [2]. With weights initialised, the samples from the data set are iterated over, the loss calculated and weights updated per batch accordingly. This was performed by first iterating through the number of batches and then individual samples of each batch. The loss is evaluated for each sample, using the function `evaluate_objective_function()`. This function also handles the process of backpropagation, evaluating the gradient of the loss with respect to changes in weights analytically, and storing these values in the weight struct of the c code. These weight structs have fields `w` and `dw`, with dimensions equal to the nodes in the previous layer and in the subsequent layer. Weights were updated after each batch had been iterated through, using the function `update_parameters()`. This function defined a double precision variable `coefficient = ((double)learning_rate/(double) batch_size)`, which was then multiplied with the accumulated gradients `dw`, to produce the second term in equation 5. weights were updated by calling the `w` field for each layer, i.e. `W_LI_L1`, `W_L1_L2`,`W_L2_L3` and `W_L3_LO`, with for loops iterating over both the neurons of the previous layer and the subsequent layer. Accumulated gradients `dw` were set to zero at the end of iteration so that gradients used for optimisation corresponded only to those of specific batches at a time. Having implemented batch SGD, the relevant files were compiled using the standard GCC compiler, using the WSL OpenSUSE Leap 15.3, on an intel i7 processor. The learning rate was set to 0.1 and the batch size to 10, with 10 epochs over which to perform the

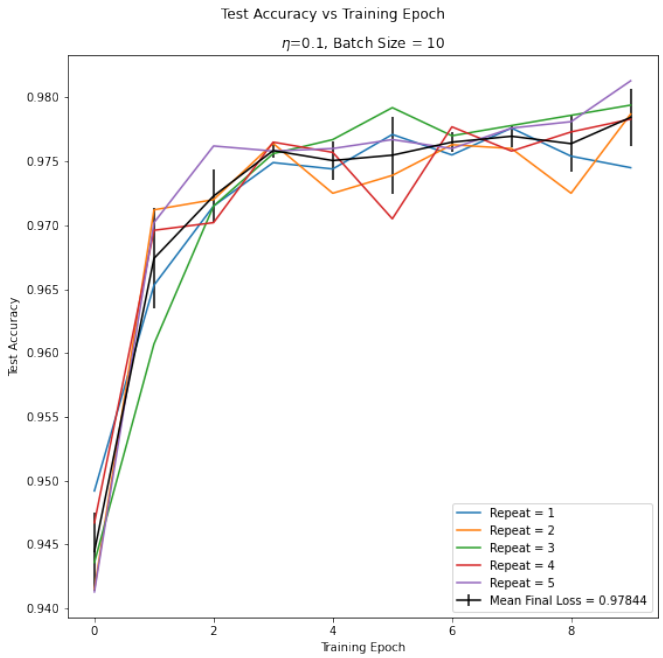


Figure 2: Convergence of artificial neural network accuracy, using batch SGD optimiser with hyperparameters: learning rate = 0.1, batch size = 10 and training epochs = 10. The ANN achieves around 97% accuracy.

training. To check the convergence of the optimiser, multiple repeats of the optimisation were performed. Figure 2 displays the results of these 5 repeats, showing test accuracy vs epoch counter on the x axis.

There is considerable variability in test accuracy from epoch 5 onwards, in particular. This may be due to SGD struggling when very close to the optimum. Generally however, the closeness of the final accuracies, within 1 % of one another, suggest that the SGD algorithm has converged. Further confidence can be placed in these results when comparing with the error rates reported by Yann LeCun et al in their work on document recognition. In figure 9 of this work, error rates are shown to vary between 0.7 and 5% for the majority of presented implementations. Furthermore, when considering similar architectures to the one used here (i.e. middle bars in the figure, with 28x28) , the error rate here of 2.16% compares well to the average of the architectures that were not fed distorted data, 3.80%. The results of their work suggest an important effect of neural network architecture on accuracy, which has not been explored here. The reader is referred to further sources, such as Bernardos and Vosniakos' 2007 paper, where it is shown that optimisation routines can even be applied to network architectures themselves to improve results [4]. The implementation itself was checked through comparison of the analytically calculated gradients with numerical, forward difference approximations. This was performed by running the optimisation with batch size of 1, and calling the numerical approximation for one weight in each layer, at every epoch. The forward difference was evaluated by calculating $\frac{f(w+\Delta w)-f(w)}{\Delta}$. The loss value for the perturbed terms were calculated using an adjusted version of `evaluate_objective_function()`, where the `store_gradient()` call and prior backward pass call had been omitted. This avoided gradients being incorrectly adjusted afterwards and affecting the numerical value being used for comparison. Also, the perturbed weight was reset to its original value – avoiding affecting subsequent gradient evaluations. Future work could simply extend this by calculating the central difference approximation for the gradients or calculating it for all weights. Alternatively, looking to the work of LeCun, a more ambitious approach would be to apply the Hessian approximation to a number of samples [5]. The numerical gradient checks can be found by running `main.c`. Using $\Delta = 0.00001$, an example value is that of the indices $i = 1, j = 1$ gradient term of `W_LI_L1`, numerically calculated to be 0.03824873, compared to the analytical value of 0.03824872.

4 2.1. Improving Convergence: Learning rate decay and Momentum

The code from the previous section was now ran with a variety of new hyperparameters, with 3 repeats each, shown in the legend of figure 3. These tests showed that 5 combinations performed far better than the others, managing to achieve just short of 98 % accuracy. The red and orange lines in figure 3 both had the fraction $\frac{\eta}{m}$, where m is

batch size, equal to 0.01. The three lower lines all had the same fraction evaluated at 0.001. While a batch size of 1, as in standard SGD, performs well and is seemingly able to avoid local minima, it is slow and requires iteration over all samples in the training set. On the other hand, batch size of 10 is still able to capture enough noise from this batch of images to move past local minima, but whilst allowing updates to be made quicker [6]. Although one would imagine a larger batch would perform best, closer resembling gradient descent - smaller batches are favourable as they allow the system to be updated and corrected more frequently. For analysis with a fixed number of epochs, comparison between different batch sizes will generally favour smaller batch sizes. Future work could consider whether training ANNs on larger batch sizes can achieve atleast as good accuracy as smaller batch sizes, if the number of updates between the two are equal as performed by Wilson and Martinez, 2003 [7].

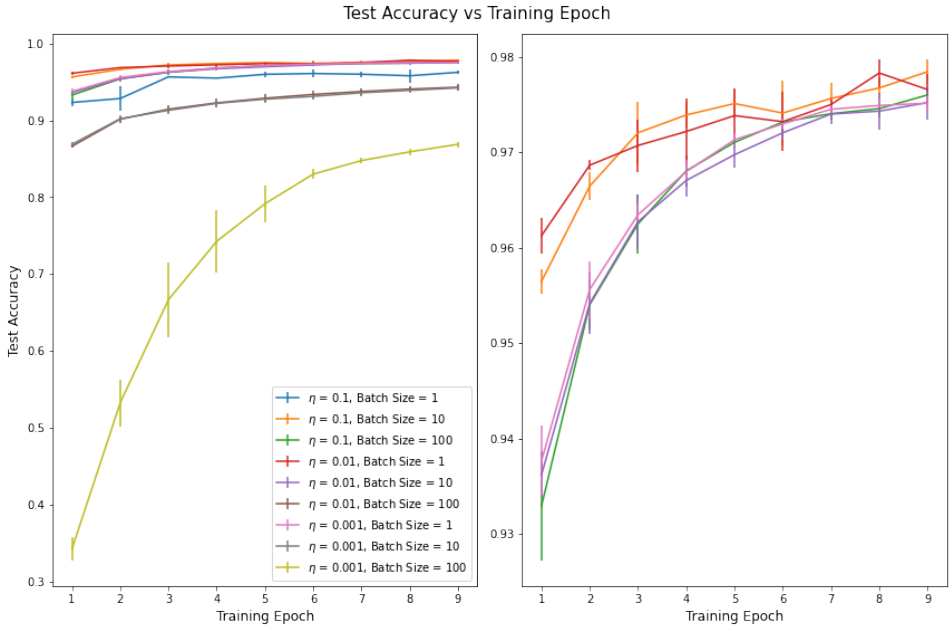


Figure 3: Convergence of artificial neural network accuracy, using batch SGD optimiser with hyperparameters as shown in legend. A second panel illustrates the best performing parameters in finer detail.

Clearly, it is beneficial for an optimiser to have a large learning rate far from a minimum, to be able to travel quickly through the parameter space; but, conversely, a small learning rate is necessary around the minimum to effect local convergence. As such, learning rate decay is introduced, updating the learning rate every epoch, k:

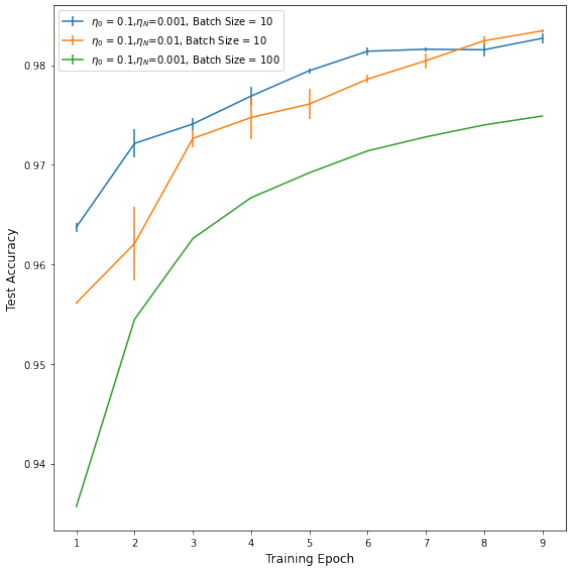
$$\eta_k = \eta_o(1-\alpha) + \alpha\eta_N, \tag{6}$$

Where α is defined as $\frac{k}{N}$ with N the total epochs. Additionally, gradient descent optimisers can become fixed in local minima, or saddle-like features. The latter leads to oscillations against steep gradients that do not actually relate to the true convergence along a different dimension. Momentum resolves this by accumulating a moving average of previous gradients:

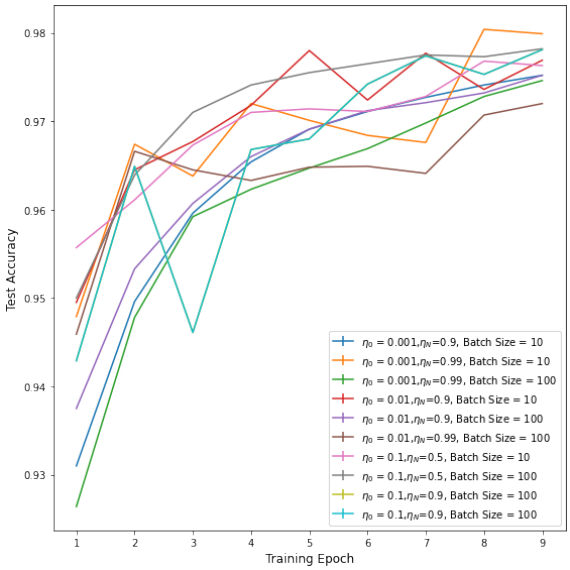
$$w = w + v = w + \beta v - \frac{\eta}{m}\sum_{i=1}^{m}\nabla L_i(x_i, w), \tag{7}$$

Where β defines the decay associated with the previous gradients. This new equation for the update of w produces larger step sizes when current gradients are parallel to previous gradients, and smaller steps where current and past gradients oppose one another, as for oscillating gradients.

In order to implement these new features, the code was expanded to accept more arguments. To avoid a panoply of redundant hyperparameters, a whole new function `initialise_optimisation_2()` was written in `optimiser.c`,



(a) Convergence of SGD optimiser employing learning rate decay, using equation 6



(b) Convergence of SGD optimiser employing momentum, using equation 7

Figure 4: Convergences of SGD when using exclusively learning rate decay or momentum

to be called by an alternative main file, `main_p2.c`. The main code was then made to request an initial and final learning rate from the user. The check on batch completion was adjusted to also diminish the learning rate, according to equation 6. The double precision variable alpha was calculated with the `epoch_counter/total_epochs`. This was performed after incrementing the epoch counter, to update following completion of the training set. On the other hand, momentum required alteration to the `neural_network.h` file. Following from equation 7, it was necessary to add a `v` field to the weight structs. Subsequently, these `v` values were updated for each connection through the new `update_parameters()` function. Also, it is important to note that the `v` fields for each connection had to be initialised to zero at the beginning of the optimisation.

First, the updated optimisation routine was ran with β set to zero. From equation 7 it follows that this produces the simpler case of SGD with learning rate decay, though potentially running slower than if learning rate decay was implemented on its own. This was initially trialled with learning rate 0.1 to 0.001 and batch size of 10, following the results shown in 3, showing batch size of 10 performing well. Three repeats were performed for this system of hyperparameters. These initial attempts already showed great results compared to not using learning rate decay, with an average test accuracy of 98%. Decay from 0.1 to 0.01 managed to achieve marginally higher results than decaying to 0.001, with almost no deviation between repeats, though it converged to an accuracy of 98% slower, shown in 4a. Perhaps having a larger learning rate at later epochs allowed the system to avoid many local minima that may be present near the true minimum. Future work could assess the effect of varying batch size as training progress. In fact, this was in the beginning stages of implementation in this work and has been left in the code, but was not yet tested, under the name `run_optimisation_4()` in `optimiser.c`. The batch size was planned to decay in conjunction with decaying learning rate, after every epoch similarly to equation 6, as figure 3 showed that learning rate of 0.01 performed best with batch size 1. Perhaps batch size could actually be increased without any decrease in learning rate to achieve the same increase in accuracy as in Smith et al, 2018 [8].

Next, however, the momentum was changed, whilst keeping learning rate constant. This did not manage to produce as accurate results as the simple case of just using learning rate decay. We see from figure 4b, that none of the momentum implementations breached 98%, even when explored over a wide range of hyperparameters. Values of $\beta = 0.5$ with learning rates of 0.1 were able to produce accuracies comparable to those with $\beta = 0.9$. Given the learning rate was already large at 0.1, it encouraged a smaller momentum coefficient to be used. Batch sizes of 1 were not used here as large variability in single samples could lead to unstable changes in weights when coupled with momentum. Future work should repeat this analysis with repeats and a smaller parameter search. Additionally, application of both learning rate decay and momentum was attempted. This surprisingly, this was found to produce very high accuracy with the hyperparameters given by: $\eta_0 = 0.1$,$\eta_N = 0.001$, $\beta = 0.9$ and $m = 10$. This is shown

in figure 5 below. Interestingly, this time we find that a batch size of 100 now performs better than batch size 10. The use of $\eta = 0.001$ was shown to converge very slowly (if at all) in figure 3, but here it is able to use larger initial steps before refining near the minimum. The larger batch size of 100 may produce less noisy gradients, which then allow the momentum to accumulate, aiding convergence. One repeat of the test with $\beta = 0.9$ and $m = 10$ produced accuracy of 98.48%, better than any values produced with only learning rate decay. As Bengio and Goodfellow explain, the learning rate is consistently one of the hardest hyperparameters to set [1]. This is because the loss function is often sensitive to specific changes in parameter space, whilst being unresponsive to changes in other parameters. It is reasonable to conclude that weight specific learning rates may have benefits, confirming what we have seen from applying the momentum here and encouraging more developed work [1].

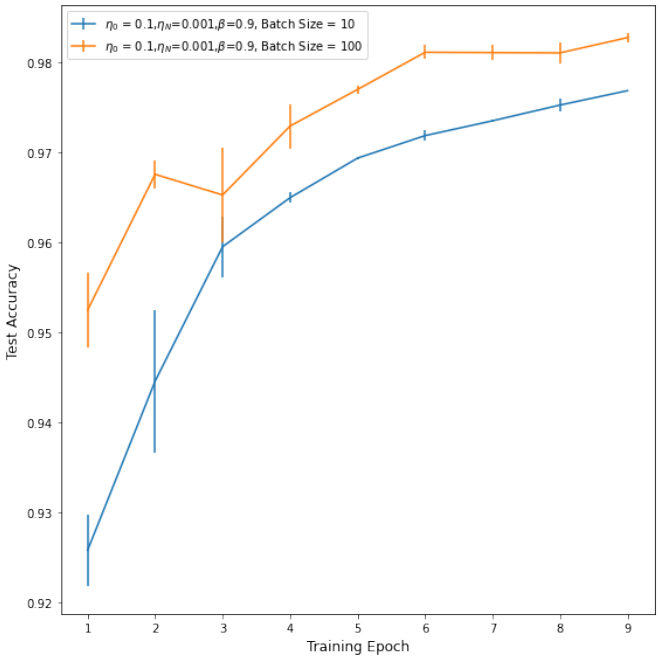


Figure 5: Convergence of artificial neural network accuracy, using batch SGD optimiser with momentum and learningr ate decay algorithm, using hyperparameters as shown in legend. Achieving above 98% accuracy.

5 Part 3: Adaptive Learning

Whilst the momentum does produce weight specific learning rates, this results in another hyperparameter to be tuned. Instead, we can explicitly tune parameters with indvidual learning rates. Adagrad is one of the simplest implementations of this, accruing the square of previous gradients. Alternatively, this history of gradients can be made to decay, as in RMSProp, to avoid computation. The two relevant equations for RMS prop are as follows:

$$p = \nu p + (1 - \nu)\nabla f(w)^2, \tag{8}$$

With p being a new feature of the connection, recording the averaged squared gradients and ν being between 0 and 1, decaying previous terms [9]. Then, the weight update:

$$w = w - \frac{\epsilon}{\sqrt{\delta + p}}\nabla f(w), \tag{9}$$

where δ is a small value used to offset the denominator to avoid division by zero and ϵ is the global learning rate. RMSprop was implemented by adding a new main file to the program, `main_p3.c`, using `run_optimisation_3()`. The weight struct in `neural_network.c` also gained a `p` field. The field `p` was initialised to 0 for all weights,

and updated with equation 9. Generally, RMSprop was very sensitive to hyperparameter setting. Many attempts were made that did not yield any reasonable results. In fact, during the process of trying to get RMSProp to work, changing the softmax function was even considered. This was applied in `neural_network.c` to try and make softmax more numerically stable to overflowing activations. This was implemented by finding the max activation value for the penultimate layer, and rescaling every activation by this value, following equation 1 from Lim and Lee, 2017 [10]. This does not change the value of softmax as the terms cancel in the fraction in equation 2. While this did coincide with the first instance of RMSprop working, this alteration to softmax was then removed when generally stable hyperparameters were found. This implementation remains in `neural_network.c` but is commented out. These first values were 0.99 and batch size of 100. With a small m and small coefficient, there can be great variability in a single batch, which can yield large values of ∇L^2, as implemented by squaring the `dw` terms in the code, which then further accumulate due to the small decay coefficient. This encourages larger batch sizes to be used, with coefficients above 0.9. For instance, batch size of 1 always resulted in numerical instability occuring. Hyperparameters around these values were then trialled, though none were able to systematically produce accuracies above 0.98, as illustrated in figure 6. Use of batch size 128 was trialled to see whether increasing this parameter slightly could improve results - $\nu = 0.99$ did produce many instances above 98% over the 3 repeats, but the convergence overall is not discernably better than the other combinations.

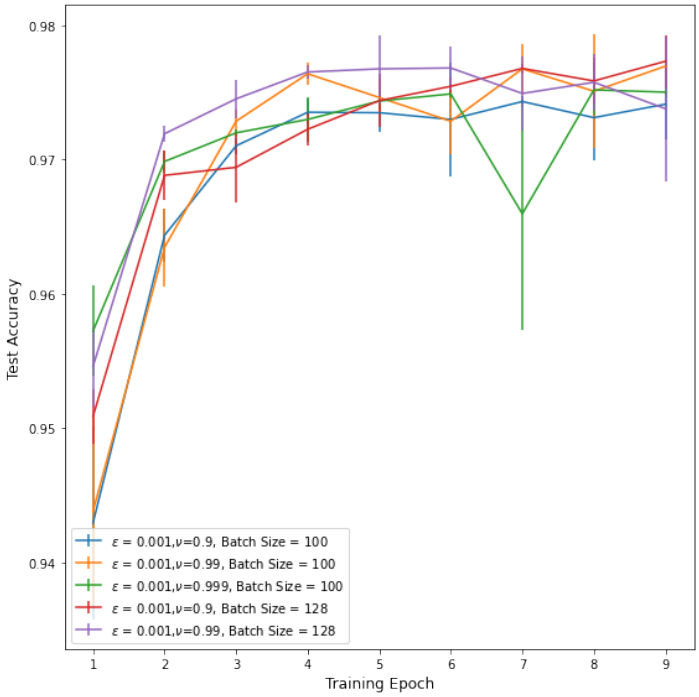


Figure 6: Convergence of artificial neural network accuracy, using batch SGD optimiser with RMSProp algorithm, using hyperparameters as shown in legend. Here we note RMSprop has failed to breach the 98% barrier, performing worse than learning rate decay, momentum or their combination.

6 Conclusion

SGD was found to be a strong optimisation method for ANN training. It was found that a batch size of 10 and learning rate of 0.1 provided the best balance of accuracy and efficiency. Furthermore, learning rate decay between 0.1 and 0.001 provided accuracies up to 98.42%. It was not possible to find a value of the momentum to recreate this accuracy, though a combination of $\beta = 0.9$, $\eta_0 = 0.1$, $\eta_N = 0.001$ and batch size 100 yielded results up to 98.48%. Finally, adding RMSprop was not found to improve results. Rather, this algorithm was very sensitive to hyper parameter selection. Immediate future work could involve implementing and evaluating the performance of the Adam optimiser. Further in the future, the role of network architecture or the implementation of batch size growth could be explored.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. The MIT Press, 2017.

[2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[3] Oliver Rhodes. Comp36212 assignment ex3: Gradient based optimisation.

[4] P.G. Benardos and G.-C. Vosniakos. Optimizing feedforward artificial neural network architecture. *Engineering Applications of Artificial Intelligence*, 20(3):365–382, 2007.

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Léon Bottou. Large-scale machine learning with stochastic gradient descent, Jan 1970.

[7] D.Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

[8] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[9] Oliver Rhodes. Comp36212 week 11: Optimisation vs training in anns.

[10] S. Lim and D. Lee. Stable improved softmax using constant normalisation. *Electronics Letters*, 53(23):1504–1506, 2017.