# Gradient-Based Optimisation

Sultan AlRashed

June 2022

## 1 Stochastic Gradient Descent

Optimizing artificial neural networks (ANNs) is a problem with many solutions over the years. However the de facto modern optimisation solution for classifying hand drawn digits is stochastic gradient descent (SGD)[13]. SGD is a stochastic approximation of an existing optimisation algorithm known as gradient descent.

SGD accomplishes this approximation by estimating the gradient from a subset of the data provided, rather than calculate the actual gradient from the entire data set. By doing this the high computational cost usually present in high-dimensional optimization problems is reduced, therefore trading a lower convergence rate for faster iterations[1].

**Definition 1** (SGD)**.**

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

*where:*

- $w$*: represents the weight of a connection.*
- $\eta$*: represents the learning rate.*
- $m$*: represents the batch size.*
- $\nabla L_i(x_i, w)$*: represents the gradient of the objective function* $L_i(x_i, w)$*, where* $L = -\sum_{k=0}^{N} \hat{y}_k \log(P_k)$ *with* $L$ *representing the loss.*

In the case of our problem domain, classifying hand drawn digits, multi-layer perceptrons (MLPs) are sufficient. Convolutional neural networks (CNNs) usually perform much better for tasks that require understanding visual signals such as RGB or gray-scale values[4][20], which includes image classification, but they present an unnecessary computational cost in the current problem domain. Even though CNNs have been shown to reach an accuracy of 99% for classifying the MNIST data set, MLPs have been shown to reach an accuracy of 97% for the same problem domain, making them more than sufficient for this use case[3].

## I Optimiser Implementation

The skeleton optimization framework present in optimiser.c was completed by implementing batch SGD. For this implementation, equation 1 was used to complete the method update_parameters(). Implementing the equation was straightforward, since many functions necessary for the functionality of this equation have already been made. The implementation was as follows:

1. Write out four double nested loops to loop over all four weight matrices in their entirety.
2. In each loop use the following code (which implements equation 1), replacing LX and LY with LI and L1 for the first set of nested loops, L1 and L2 for the second set of nested loops and so on:

```
w_LX_LY[i][j].w = w_LX_LY[i][j].w - ((learning_rate/batch_size) * w_LX_LY[i][j].dw
    );
w_LX_LY[i][j].dw = 0.0;
```

   Here dw represents the gradient of the objective function, while w represents the weight.

## II Validation through Numerical Differentiation

By using forward difference approximation, a numerical differentiation technique, we can approximate the gradient subject to the following equation:

**Definition 2** (Forward Difference Approximation of Gradient)**.**

$$\nabla L(x, w) = \frac{L(x, w + \epsilon) - L(x, w)}{\epsilon} \tag{2}$$

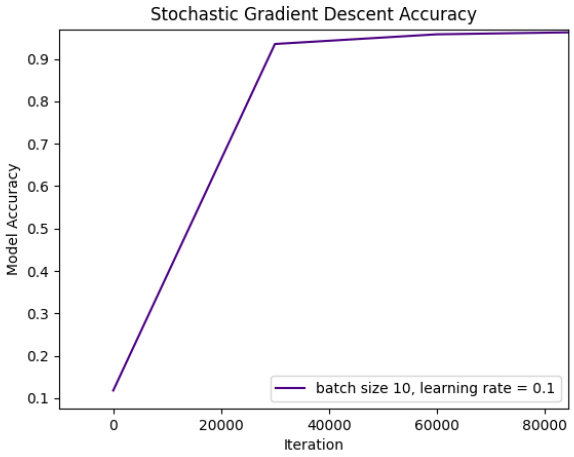
*Where:*

- $\epsilon$*: represents the step size.*

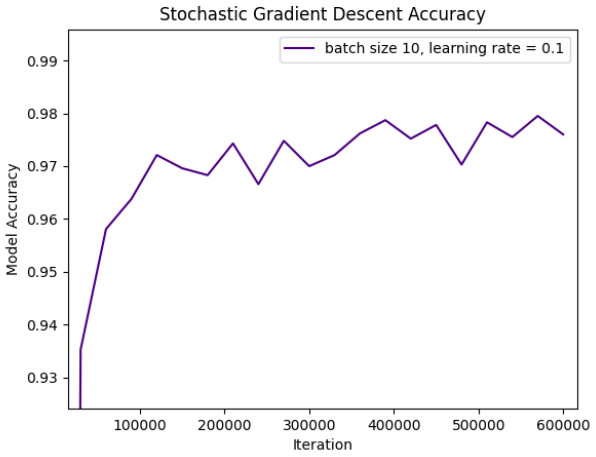
Therefore we can actualize this in software by creating a new function in optimiser.c that we will call numeric_diff(). In the function we calculate the loss currently using the two functions evaluate_forward_pass() and compute_xent_loss(). Afterwards we adjust the weight of one weight connection in W_La_Lb (where a and b are two connected layers) by our chosen step size and calculate the loss again using the same two functions. Finally we apply the above equation to get the forward difference approximation of the gradient and compare it against the actual gradient stored in W_La_Lb[x][y].dw. To ensure the analytical solution's gradient is stored correctly we also must run evaluate_backward_pass_sparse() and evaluate_weight_updates() both. At last we can now compare the two values and determine the validity of the analytical solution.

After repeated testing the error margin was found to be at most $\approx 10\%$, but that could also be explained by the fact that some extra tweaking with the step size was necessary as both the step size being too small or too large can cause huge differences. In terms of compute time numerical calculation of the gradient was much faster, cutting compute time by half in some cases.

## III Experimental Results




(a) Figure shows initial convergence.




(b) Figure shows finer details near the optimum.

Within 20,000 iterations the model begins to reach an accuracy above 90%, which is an impressive initial result that occurred without much tweaking on the base SGD algorithm. However it is important to bear in mind that the MNIST data set is not a particularly difficult data set for neural networks.

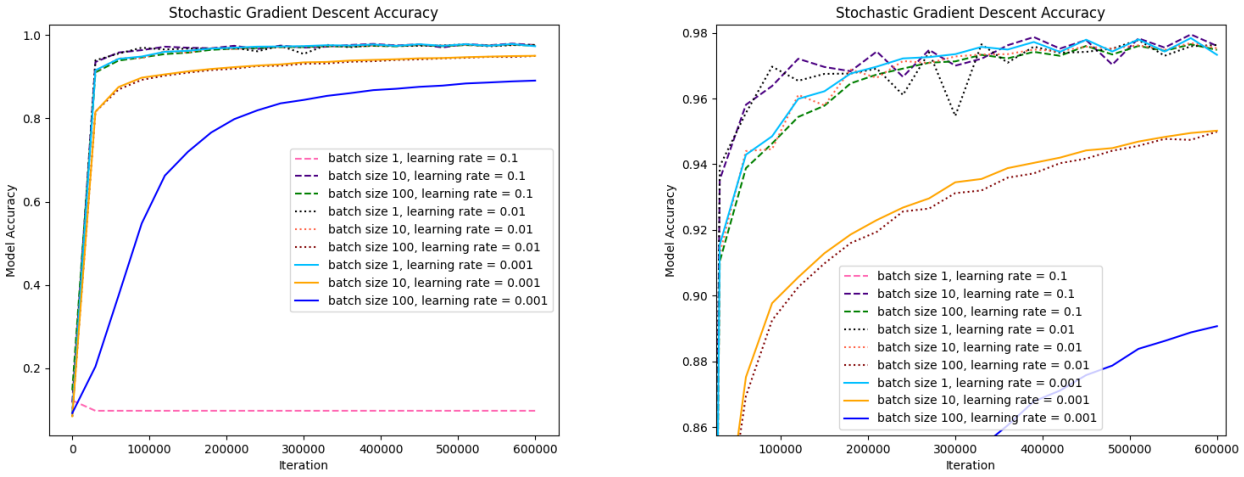
The question of whether the model has found an optimum is an interesting question, since this definition is subjective to the literature surrounding the problem domain more so than with a definite mathematical one. This model with its current hyperparameters have failed to reach an optimum for two central reasons, the first being that it has been shown that MLPs can reach much better accuracy than the maximum of $\approx 97.6\%$ present here for the MNIST data set[12]. Secondly it has been shown that convolutional neural networks can reach an accuracy above even 99% for the MNIST data set[3].

With these two points in mind one can conclude that firstly the hyperparameters used in this experiment are sub-optimal and secondly the architecture itself is sub-optimal, therefore the model has not reached an optimum.

# 2 Improving Convergence

Improving the convergence of SGD can be accomplished by adding some extensions to the base algorithm, possible extensions include: implicit updates, averaging, learning rate decay, and momentum based weight updates but this section will focus on the latter two. Initially though, we will investigate the effect of batch size and learning rate on accuracy for a better understanding of the hyperparameters at play.

## I Impact of Batch Size and Learning Rate




(a) Accuracy of different batch sizes and learning rates tested.

(b) Same as figure (a), but with a zoom to show finer details near the optimum.

An interesting result can be seen where that when $\frac{\text{Batch Size}}{\text{Learning Rate}} \geq 10000$, the model takes much longer to converge to the optimum. When the same ratio is $\frac{\text{Batch Size}}{\text{Learning Rate}} \leq 10$, the model is completely unable to converge and fails to generalize.

A few papers have come out arguing that SGD function better on a wide minima[7], while others proved that sharper minima leads to worse generalisation performance for SGD[10]. Taking this into account in conjunction with the results shown above, one could posit that the ratio of learning rate to batch size directly correlates to SGD's ability to converge to a wider minima[9], with each value on its own not having as much of an effect. In fact this has been proven empirically in a paper[9][2][5], where it was shown that this ratio directly determines SGD's ability to generalize and converge to an optimum.

## II Learning Rate Decay

The learning rate as a hyperparameter has been shown to pose serious problems to the convergence of SGD if not set correctly. If the learning rate is too low convergence becomes slow, while if it is too high the algorithm begins to diverge. Learning rate decay is a simple band-aid to this issue, by offering a high learning rate initially that decays into a low learning rate at the end we can ensure rapid initial convergence, and local refinement close to the optimum[23].

**Definition 3** (Learning Rate Decay)**.**

$$\eta_k = \eta_0(1-\alpha) + \alpha\eta_N \;\; where \; \alpha = \frac{k}{N} \tag{3}$$

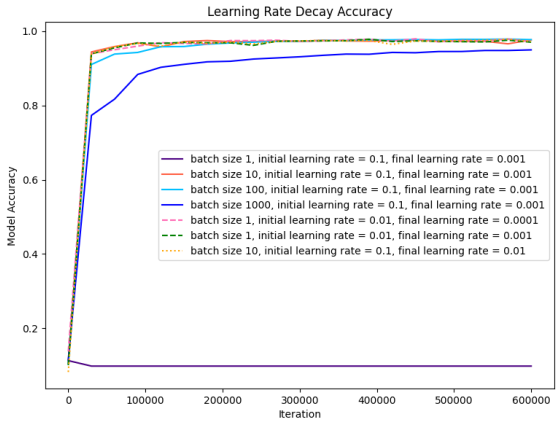
However it has been shown before that increasing the batch size during training achieves the same result, while allowing for better parallelism and a potential massive reduction in training time, it is the superior alternative to learning rate decay[16]. This method will be explored in conjunction with learning rate decay to reproduce the results found in the paper this was discussed.

Implementing linear learning rate decay was straightforward:

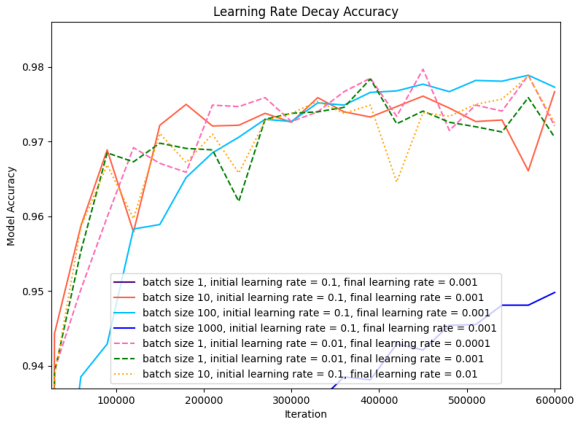
1. New arguments are taken from the user upon program execution, the initial learning rate and the final learning rate. They are both applied as global variables in optimiser.c afterwards.

2. A new function was created to handle the implementation, for cleanliness. Inside the function the global learning rate variable was changed in accordance with equation 3.

3. The function is called just before running any form of weight update, in run_optimisation().




(a) Accuracy of different batch sizes, initial learning rates and final learning rates tested.

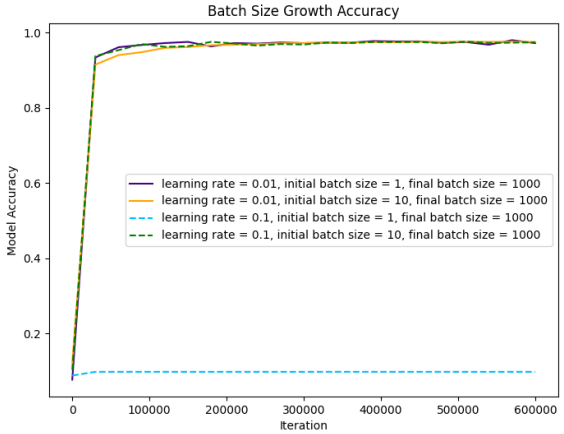



(b) Same as figure (a), but with a zoom to show finer details near the optimum.

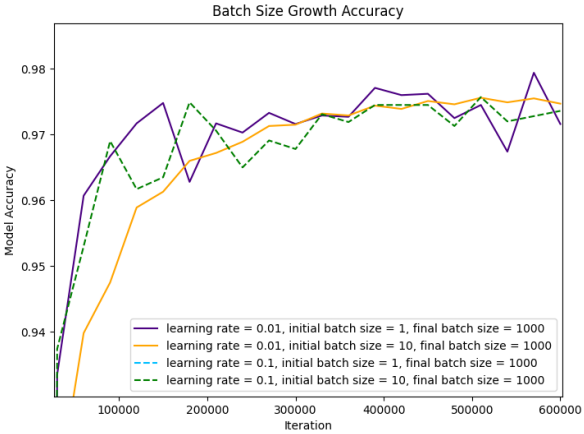
When compared to normal SGD learning rate decay shows more stability towards the optimum while also providing rapid initial convergence. This makes sense since learning rate governs how malleable the model is to changes in weight, by reducing the learning rate as we go further into the iterations the model now has to refine around the optimum rather than oscillate back and forth between local minima and maxima (as can be seen in the SGD figures).

## III Batch Size Growth

Batch size growth is implemented in the exact same way as learning rate decay, even replicating the same equation, with the only different being that we replace learning rate, initial learning rate, and final learning rate with batch size, initial batch size, and final batch size respectively.




(a) Accuracy of different learning rates, initial batch sizes and final batch sizes tested.




(b) Same as figure (a), but with a zoom to show finer details near the optimum.

As the paper suggested[16], scaling the batch size up presents very similar performance to decaying the learning rate. A great boon however is that by increasing the batch size we increase the step size which in

turn reduces the number of parameters updated while training a model[5], this offers a respectable reduction in computation time while also using these large batches across machines through parallelism[8]. Batch size growth seems to be the superior alternative when put beside learning rate decay, however for the sake of the assignment learning rate decay will remain the focal point of this section alongside momentum based weight updates.

## IV Momentum Based Weight Updates

One can think of momentum in SGD as a ball rolling down a hill, accelerating learning at times where the model might be stuck in a local minima. In fact the model being stuck can be seen in figure 2(b) where certain lines oscillate back and forth in regions of pathological curvature.

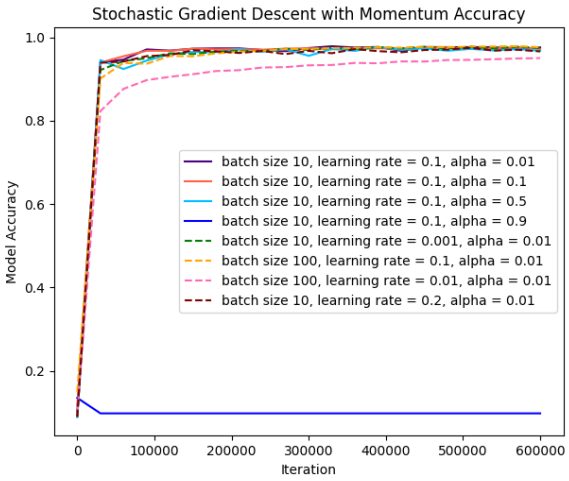
**Definition 4** (Momentum Based Weight Updates)**.**

$$\begin{aligned} v &= \alpha v - \frac{\eta}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w) \\ w &= w + v \end{aligned} \tag{4}$$

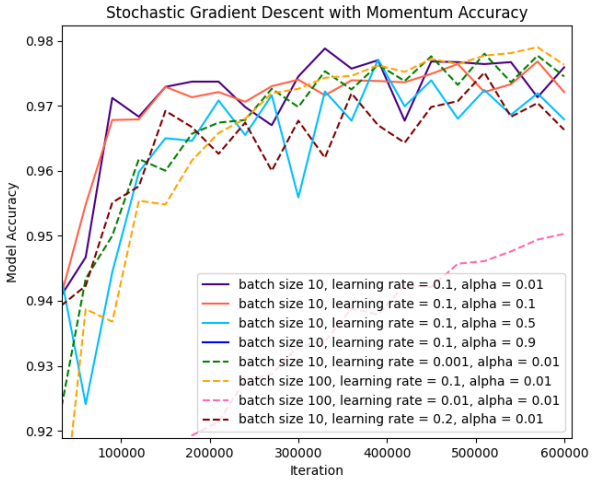
Implementing momentum based weight updates required a bit more tinkering around with the internal code, and was accomplished as follows:

1. The weight structure weight_struct_t was modified by adding an additional double variable called v, representing the velocity.
2. The new variable v was initialised to 0 in the initialise_weight_matrices() function present in neural_networks.c.
3. An implementation similar to that of the original update_parameters() was performed, with four double nested loops that loop over each possible connection in the neural network, by accessing the weight matrices, being created.
4. In each loop, the momentum based weight update equation was implemented.

The results of many experiments were as follows:




(a) Accuracy of different batch sizes, decay rates $\alpha$, and learning rates tested.

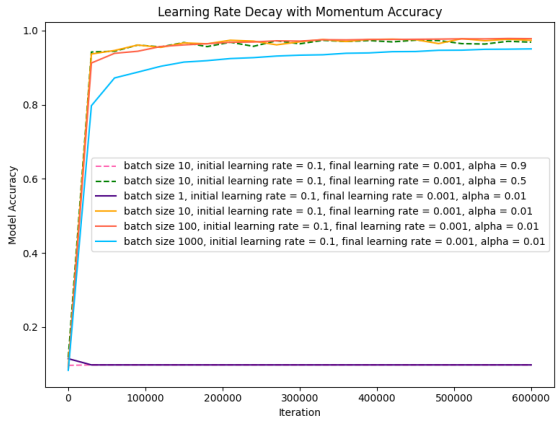



(b) Same as figure (a), but with a zoom to show finer details near the optimum.

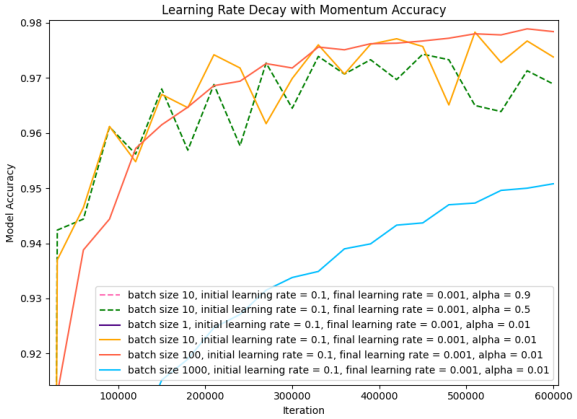
From this one can notice that ideally the decay rate should be quite low, the blue solid line with a decay rate of 0.9 was unable to generalize or converge at all and failed. Generally when the decay rate was kept low, stability was far better than that of SGD's. Even though in the zoomed in graph it might seem that there are a lot of oscillations, but this is due to the extra fine zoom given to notice these minor changes.

This improvement in stability is in line with the current consensus that adding momentum to SGD overall improves stability foremost while also providing better results[14][18].

## V Learning Rate Decay with Momentum




(a) Accuracy of different batch sizes and decay rates with best initial learning rates and final learning rates tested.




(b) Same as figure (a), but with a zoom to show finer details near the optimum.

When combining both learning rate decay and momentum together, the model exhibits both the rapid initial convergence typical in learning rate decay and the stability typical in momentum. In fact the bright red solid line seems to have barely any oscillations throughout, however it does not seem that an optimum was reached yet. Maybe if the model was allowed to run for longer however an optimum might be in reach.

The ideal hyperparameters from the testing above can be seen to be batch size = 100, initial learning rate = 0.1, final learning rate = 0.001, $\alpha = 0.01$. However the lack of testing different learning rate combinations may have prevented a better set of hyperparameters from appearing, the justification for not changing them however is their dominant performance when testing learning rate decay in isolation.

The MNIST data set is a popular problem domain, time and time again the combination of learning rate decay and momentum has shown great results. Therefore this performance is expected and corroborated by papers[15][17][18].

# 3 Adaptive Learning

The learning rate is a finicky hyperparameter that can lead to a high degree of variation in results, with the learning rate being such an important hyperparameter to optimise for many people have come up with innovative solutions to adaptive select for the ideal learning rate throughout the training process. The adaptive learning rate implementation this section will focus on will be Adam, which stands for adaptive moment estimation[11].

Adam has been shown to reach an accuracy of 99.12% on the MNIST data set when using a convolutional neural network[19], quite an impressive result that exceeds that of SGD's. Adam is also a straightforward algorithm to implement that is computationally efficient, has low memory requirements, and is well suited for large data sets[11]. Another benefit is that the hyperparameters posses intuitive meaning behind them, allowing for relatively little hyperparameter tuning.

Comparatively to RMSprop and Adagrad, Adam is much more computationally efficient to implement, while still presenting an accuracy that can exceed 99% which is more than enough for our use case.

Due to all of these factors, Adam was chosen as the adaptive learning algorithm to explore.

## I Adam

In an abstract sense if one thinks of SGD with momentum as a ball running down a slope, one can imagine Adam works like a heavy ball that has friction. In a more theoretical sense, Adam prefers flat minima in the error surface[6].

Adam works by keeping an exponentially decaying average of past gradients $m_t$ (similar to momentum) while also storing an exponentially decaying average of past squared gradients $v_t$. To compute these variables, the following functions can be used:

**Definition 5** (Adam First and Second Moment)**.**

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{5}$$

*where:*

- $m_t$*: represents the estimate of the first moment, or the mean.*
- $v_t$*: represents the estimate of the second moment, or the uncentered variance.*
- $\beta_1$ *and* $\beta_2$*: represent decay rates, which are hyperparameters.*
- $g_t$*: represents the objective function.*

However the authors of the paper where Adam was introduced noticed that both $m_t$ and $v_t$ tend to be biased towards 0 since they are initialized as vectors of 0s[11], this effect is more pronounced when the decay rates are small. To counteract these biases, one must compute the bias-corrected versions of $m_t$ and $v_t$:

**Definition 6** (Adam Bias Correction)**.**

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{6}$$

Finally these variables are used to update the weight parameters present in our multi-layer perceptron:

**Definition 7** (Adam Update Rule)**.**

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{7}$$

The authors proposed the default values of 0.9, 0.999, and $10^{-8}$ for $\beta_1$, $\beta_2$, and $\epsilon$ respectively which will be used as the baseline for hyperparameter tuning.

# II Software Implementation

Implementing Adam was done by using the original implementation of SGD in update_parameters() as a baseline and then following was done:

1. $\beta_1$, $\beta_2$, and $\epsilon$ are defined as doubles with values of 0.9, 0.999, and $10^{-8}$.
2. Four double nested loops are initialized with each loop being responsible for one of the four weight matrices, looping over its entirety.
3. In each loop we calculate $m_t$, the mean, and $v_t$, the variance, by applying the first and second moment equations in equation 7.
4. In each loop we further calculate the bias corrected versions of the first and second moment to use in the update rule.
5. Finally in each loop we use the update rule provided in equation 7 to update the weight, and set the gradient back to 0.

## III Experimental Results

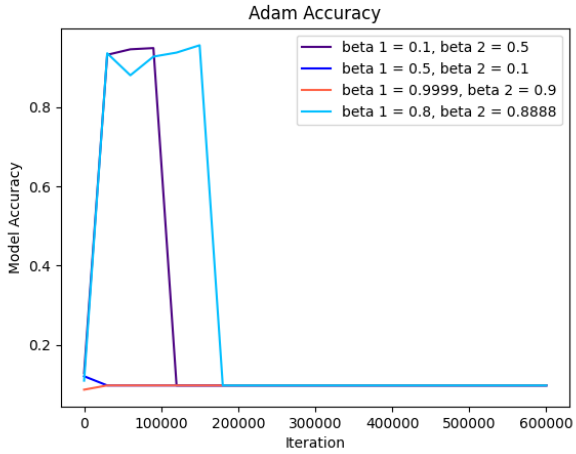
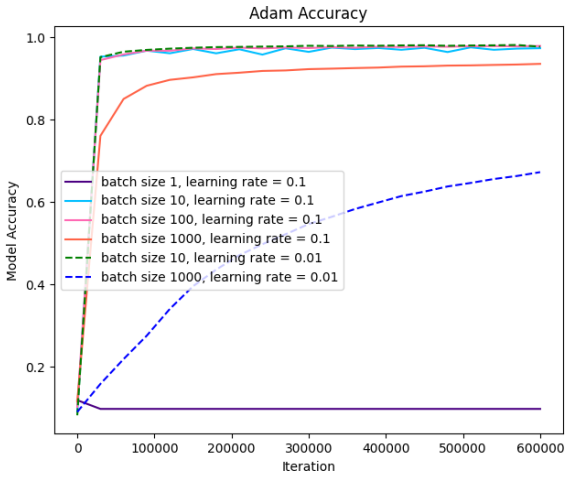
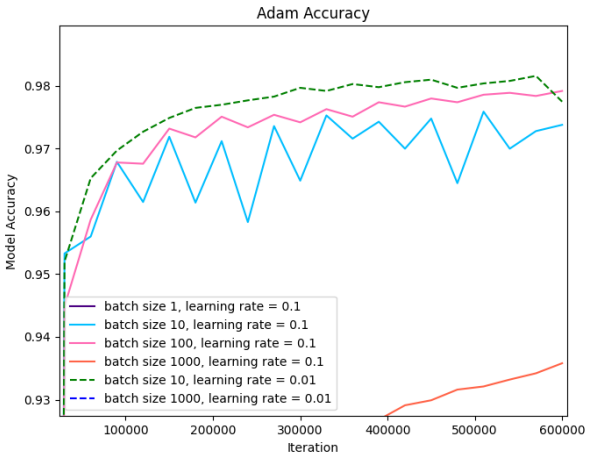



Figure 7: Accuracy of different values for $\beta_1$ and $\beta_2$, with learning rate and batch size fixed.

Clearly the default hyperparameters for $\beta_1$ and $\beta_2$ suggested by the authors of Adam are the ideal to get Adam working, as none of the ones experimented with here seem to converge at all.




(a) Accuracy of different batch sizes and learning rates.




(b) Same as figure (a), but with a zoom to show finer details near the optimum.

The dashed maroon line here shows massive stability, in fact even when we zoom in closely it seems that the line does not oscillate back and forth much at all. This is a testament to the benefits of adaptive learning rates. The results here corroborate that of many studies done on Adam, where Adam shows stability, rapid initial convergence, and great yields in accuracy[21][22]. By also reducing the importance of hyperparameters, Adam makes training a neural network a much less painful process of trial and error.

# 4 Conclusion

All in all it is clear that Adam provides the best results with the lowest computational effort and hyperparameter tuning. However using SGD in combination with learning rate decay (or maybe batch size growth instead) and momentum provides fierce competition, even matching Adam in stability for certain hyperparameters. Many things could have been done better in this report, running these experiments for longer than 10 epochs each will surely generate very interesting results but is too computationally demanding for my machine to work with and keep on reliably for long periods of time.

# References


[1] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20, 2007.

[2] Thomas M Breuel. The effects of hyperparameters on sgd training of neural networks. *arXiv preprint arXiv:1508.02788*, 2015.

[3] Karishma Dasgaonkar and Swati Chopade. Analysis of multi-layered perceptron, radial basis function and convolutional neural networks in recognizing handwritten digits. *International Journal of Advance Research, Ideas and Innovations in Technology*, 4(3):2429–2431, 2018.

[4] Meha Desai and Manan Shah. An anatomization on breast cancer detection and diagnosis employing multi-layer perceptron neural network (mlp) and convolutional neural network (cnn). *Clinical eHealth*, 4:1–11, 2021.

[5] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[6] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural computation*, 9(1):1–42, 1997.

[8] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in neural information processing systems*, 30, 2017.

[9] Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[10] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[13] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[14] Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum. *Advances in Neural Information Processing Systems*, 33:18261–18271, 2020.

[15] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[16] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[17] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, ON, Canada, 2013.

[18] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[19] Ange Tato and Roger Nkambou. Improving adam optimizer. 2018.

[20] Shamik Tiwari. Dermatoscopy using multi-layer perceptron, convolution neural network, and capsule network to differentiate malignant melanoma from benign nevus. *International Journal of Healthcare Information Systems and Informatics (IJHISI)*, 16(3):58–73, 2021.

[21] Sho Yaida. Fluctuation-dissipation relations for stochastic gradient descent. *arXiv preprint arXiv:1810.00004*, 2018.

[22] Kazunori D Yamada. Hyperparameter-free optimizer of stochastic gradient descent that incorporates unit correction and moment estimation. *BioRxiv*, page 348557, 2018.

[23] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I Jordan. How does learning rate decay help modern neural networks? *arXiv preprint arXiv:1908.01878*, 2019.