Training Neural Networks: Evolutionary Algorithms and Gradient Descent

Adeline Southard

1 Problem Statement

The goal of this project is to explore the performance of neural networks that use evolutionary algorithms to train their weights relative to gradient descent based method(s). Gradient descent methods have a more efficient convergence to optima in a fitness landscape, but they are highly prone to converging to local minima/maxima. However, evolutionary algorithms can be designed to perform better in rugged fitness landscapes at the cost of converging more slowly.

To accomplish this, I will implement a gradient based optimization method alongside the evolutionary algorithm based method, similar to what we have already written for class, and compare the performance of both neural networks and compare the performances after training with the two methods. I will also look at convergence time of both methods, and explore the network performance, i.e. compare faster converging gradient based methods (more prone to getting stuck in local optima) with slower converging evolutionary algorithm based methods (less prone to getting stuck in local optima). Neural networks are known to have local optima. Perhaps evolutionary methods can find a better optima than gradient descent based methods.

Since we have already experimented with the MNIST dataset in class, I will use that for training my neural networks.

2 Related Work

The basis and major influence for this project was the paper Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning by Such et al. [2]. This paper discusses the use of the same genetic algorithm I use in this project (SimpleGA) on reinforcement learning problems and compares them to gradient descent-based methods. They find that in some cases, the genetic algorithm is, as the title implies, competitive in terms of performance. This piqued my interest, and became the basis for my research questions and problem statement of wanting to do my own gradient descent implementation and compare the performance against a dataset we had already used in class (MNIST). Note that the MNIST image classification problem differs significantly from the problems used in the paper, as will become relevant later.

Looking further, the use of genetic algorithms for training neural networks are also described in Ahmad et al. [1], Morse et al. [3], Pendharkar [4]. Use of a combination of gradient descent and genetic algorithms is discussed in Xue et al. [6]. These papers helped further inform and guide the project.

Finally, the implementation of the gradient descent algorithm was helped by Deep Learning by Goodfellow et al. [2].

3 Methodology

3.1 Data Preparation

For this project, I use the MNIST dataset to explore the use of gradient descent and evolutionary methods of training. The MNIST dataset consists of 70,000 images of handwritten digits from 0-9. The dataset is split into 60,000 training images and 10,000 test images. The images are 28x28 pixels, where each pixel value is between 0 and 255. The labels are numbers between 0 and 9. I do not deskew (straighten tilted images), with the intent being to leave the images as-is for the Neural Network (NN) training and testing. Since the NNs that I will be exploring in this project have over 10x the number of parameters (over 10x the genome length) that we used in the homework 6, I expect the NN to be able to handle skewed images and still perform well.

Due to memory limitations of my GPU, and the larger NN size, I downscale the images from 28x28 to 14x14 to keep the genome size down for the evolutionary methods (this is not neccesary for the gradient descent methods, but I did not want to treat the image resolution as a parameter for comparison here). For the downsampling I just use the average of 2x2 pixel blocks.

To prepare the data for training, testing and validation, I split the data into train, test, and validation sets. Since the MNIST dataset is already split into training and testing sets, I only need to split the training set into training and validation sets. I use the same number of verification images as the test set, which is 10,000 images. Next, I flatten the images to a 1D arrays since this is what will be used as input to the NN.

As a last step I normalize the pixel values to have zero mean and unit variance. This is a common preprocessing step for training NNs. The mean and standard deviation are calculated only on the training set and then applied to the test and validation sets. This is to avoid the possibility of information propagating from the test and validation sets into the training set. Another way to think about this is that the training set is the only set that the model has seen, so the mean and standard deviation should be calculated only on this set and then applied to the other sets of images for which we would not have the mean and standard deviation ahead of time.

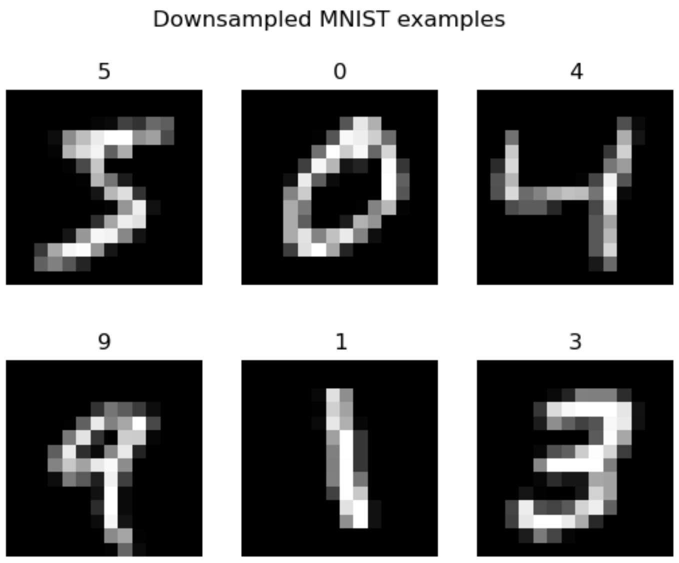


Figure 1. Examples of a few random MNIST digits after downsampling. Downsampling was performed by averaging 2x2 pixel blocks, then normalizing the pixel values to have zero mean and unit variance.

3.2 Neural Network Architecture

For the single layer NN used in homework 6, the output of the single layer NN was just a linear combination of the inputs. That is, a 10x196 matrix of weights multiplied the input flattend image pixel values (196 pixels) resulting in an output vector of length 10. We then took the argmax of the output vector as the predicted one-hot encoded label and optimized the weights to maximize the accuracy of label prediction for the training images. This argmax operation is an example of an activation function, however it is not a differentiable activation function. In general, an activation function is a function that takes the output of a layer of a neural network and transforms it for input to the next layer of the network.

Instead of using argmax, which is not differentiable, another approach is to use what is called the softmax function. The softmax function is a differentiable activation function, which is important when using gradient descent methods that require derivatives exist. The softmax function takes a vector of real numbers $\mathbf{x}$ and returns a vector of the same length $\mathbf{f}(\mathbf{x})$, where the ith element $\mathbf{f}(\mathbf{x})_i$ is in the range (0, 1), and the sum of the elements $\mathbf{f}(\mathbf{x})_i$ is 1. The softmax function is defined as:

$$\mathbf{f}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The output of the softmax function is correctly normalized to be interpreted like a probability that a label belongs to a given image. For the last layer of the network, I will use the softmax activation function since it provides a probability like result for each label that can be easily interpreted.

For the hidden layers of the NN, I first used another type of activation function called the Rectified Linear Unit, or ReLU, activation function. The ReLU function is popular, in part since it provides efficient gradient calculation (derivative is just 0 or 1). Also, the ReLU adds non-linearity to the network, which is required for the network to be able to represent a complex function. In general, at least some activation functions used in a NN must be non-linear for the NN to work, since if they were all linear then the entire multi-layer network would be a linear function of the input, so the multi-layer NN would just boil down to a matrix times the input just like a single layer network! The non-linearity of the activation functions is what enables NNs to be able to represent highly complex functions. Without nonlinearity of activation functions the output of the network would just be a matrix times the input (plus a constant bias vector), regardless of the number of layers, which is a very simple (and likely not representative) function that severly limits the predictive capability of the NN.

The ReLU function is defined as:

$$f(x) = max(0, x)$$

While this activation function worked reasonable well, I did encounter some issues when using multiple hidden layers with the ReLU activation function. The issue is that the ReLU function can cause the output of a neuron to be zero, which can cause the gradient to be zero, which can cause the weights to not be updated and the optimization gets stuck at what amounts to a local minimum with bad performance. After some research I found that this is called the dying ReLU problem. To address this issue, the Leaky ReLU activation function can be used, which is defined as:

$$f(x) = max(0, x) + \alpha\ min(0, x)$$

where α is a small constant, typically 0.01, which is what I use here. The Leaky ReLU function is similar to the ReLU function, but it allows a small gradient when the input is less than zero, which can help prevent the gradient from becoming zero and the optimization from getting stuck.

A similar issue can occur when using the softmax function, which has a sigmoid shape, and can cause the gradient to be zero when the input is very large or very small since the function saturates at 0 and 1, respectively. To address this issue, the logarithm of the softmax function can be used in the loss function, which essentially undoes the exponential operation of the softmax function and can help prevent the gradient from becoming zero. The logarithm of the softmax function can be expressed as:

$$f(x) = log(\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}) = x_i - log(\sum_{j=1}^{n} e^{x_j})$$

The first term in the expression shows that the input x_i always has a contribution to the gradient since it never saturates, which can help prevent the gradient from becoming zero. Avoiding the gradient from becoming zero is a very important consideration when designing a NN loss function [2]. Because the log function is always increasing (monotonicly increasing function), the argmax of the softmax function is the same as the argmax of the log softmax function, so the log softmax function can be used in place of the softmax function in the loss function.

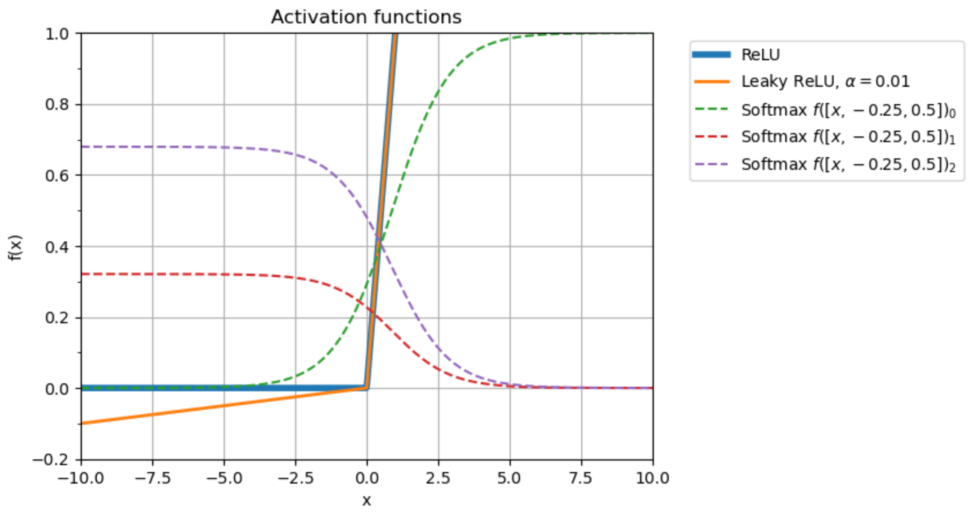


Figure 2. Demonstration of behavior of the ReLU, Leaky ReLU, and Softmax functions.

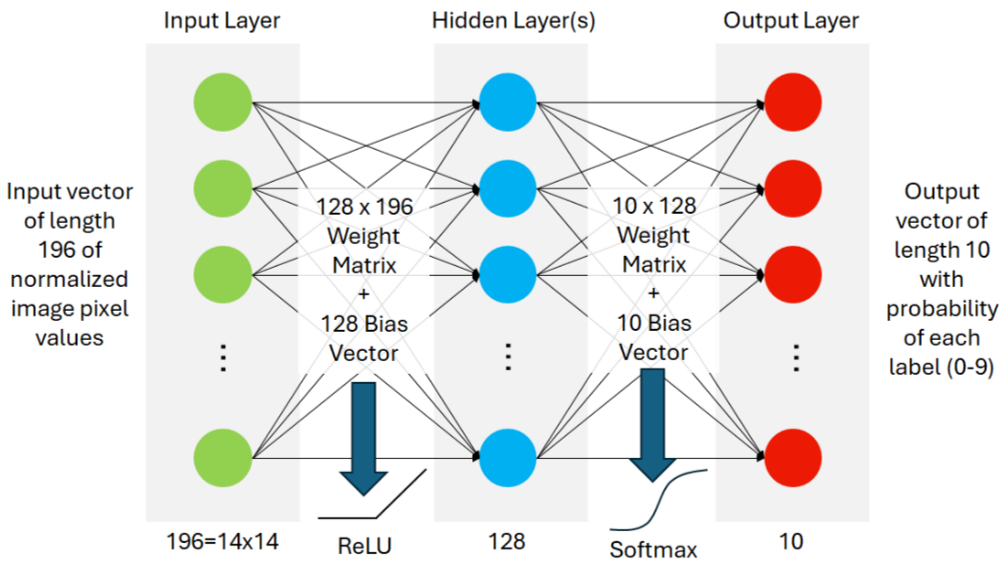

Input Layer
Hidden Layer(s)
Output Layer
Input vector of length 196 of normalized image pixel values
128 x 196 Weight Matrix + 128 Bias Vector
10 x 128 Weight Matrix + 10 Bias Vector
Output vector of length 10 with probability of each label (0-9)
196=14x14
ReLU
128
Softmax
10

Figure 3. Diagram representing the neural network architecture used to train on the MNIST dataset.

The NN prediction function returns the predicted one-hot encoded label given network parameters (weights and biases) and a flattened image from which to infer the label. This operation is called inference. The larger the network, the more computationally expensive the inference operation becomes (and hence the more CO2 emissions). While training the network can be computationally expensive, it only needs to be done once, whereas inference happens very often. Hence it is important to minimize the size of the network while maintaining high accuracy, which is an important aspect of this project.

The network parameters are organized in a list with length equal to the number of layers. The list contains tuples of parameters for each layer. Each layer parameter tuple contains the weight matrix and bias vector.

3.3 Loss Function

The loss function is defined as the negative of the mean log probability (i.e. the log of the output from the softmax activation function for the last layer) of the target labels. If the mean log probability predicted by the network is high for the target labels, then the network is working well (ideally the predicted probability of the target label is 1, so the -log of this is 0). Note that since the probability is bounded between 0 and 1, the loss function (negative of the mean of the log probability) is between 0 and ∞, and also is differentiable since we are using a softmax probability rather than argmax like the accuracy function.

Because the softmax activation function used for the last layer of the network always rescales everything to a probability between zero and 1, the output of the network is not really a function of how large or small the inputs to the network, or the network weights and biases, are (it is more a function of the relative difference of the parameters). Because of this I noticed that the network parameters can float around during optimization and things did not give good coverage in some cases. After doing some research on the topic this is not unusual and can be remedied by using a technique called regularization. To do this, I add a regularization term to the loss function that penalizes large weights. This is done by adding the sum of the squares of the weights to the loss function. The regularization term is multiplied by a hyperparameter reg_fac that controls the strength of the regularization.

The regularization term also helps to prevent overfitting. With a high penalization on the size of (sum of squares) the network parameters, the network is more likely to focus on the most important features of the data using fewer (in a sum squared sense) parameters, since more parameters would increase the regularization term.

To calculate the gradient of the loss function with respect to the neural network parameters, I will use the JAX grad function. There is a lot going on behind the curtain here, but what JAX does is track all the operations to calculate the loss (computational graph), and works backwards (back propagation) using the chain rule.

The JAX grad function takes a function as input (in this case loss) and returns a function that calculates the gradient of the input loss function with respect to its first argument.

3.4 Stochastic Gradient Descent With Momentum (SGDM)

3.4.1 SGDM Justification

For training of the network I use stochastic gradient descent with momentum (SGDM), since it is a simple and effective optimization algorithm. With tuning of the learning rate and momentum, it can be very effective, often outperforming slightly more complex optimization algorithms such as the popular Adam method, which is another type of stochastic gradient descent method. Without tuning, Adam is typically more robust, but because in this project we are comparing the 'best case' training performance (with hyperparameter tuning), I choose to use SGDM as it is often the most performant in this case.

I wrote my own implementation of SGDM, since a major part of this project was to learn about gradient descent methods. I implemented SGDM using JAX, which took significant effort since I had to learn how to use the JAX library in addition to SGD methods. However, the effort was worth it since JAX dramatically improved (by orders of magnitude) the training time by leveraging my GPU.

3.4.2 Data Batching

The concept of batching is important for training large datasets with SGDM (and other methods). The training data is divided into smaller batches, and the network parameters are updated using each batch in turn until the whole data set is used. One cycle through all the batches is called an epoch. Using batches can reduce memory use and significantly speed up training (it takes much less computation to calculate the loss and loss gradients with smaller batches of images). The batch size is a hyperparameter that can be tuned, with typical values being 128 and 256. For each epoch, I randomly shuffle the batches in the training data to ensure that the network is not biased by the order of the data.

Note that the derivatives (gradient) of the NN parameters with respect to the loss function are calculated using the batch of data, not the entire training set. Hence, for each batch the gradient will be slightly (or perhaps even significantly) different. By using batches, diversity of the gradients is introduce and can help avoid local minima. This is a key part of stochastic gradient descent (SGD), since the gradient is calculated using a stochastic (random) sample of and/or a random shuffling of fixed batches of the data. The stochastic nature of the gradient calculation can help the optimization algorithm escape local minima, and can also help with generalization (reducing overfitting) since the network is not overfitting to the entire training set.

3.4.3 Update Function

The update function for SGD with momentum is defined as:

$$v_{t+1} = \rho v_t + \nabla L(\theta_t)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

where:

- v_t is analogous to velocity at time t
- ρ is analogous to mass (if $\rho = 0$ then there is no momentum, and the update is just gradient descent w/o momentum)
- α is the learning rate
- $\nabla L(\theta_t)$ is the gradient, which is a vector of derivatives of the loss function w.r.t. the network parameters at time t
- θ_t is a vector of the network parameters at time t

The idea behind adding momentum is to smooth out the updates over time and to help the optimizer escape local minima. The momentum term is a weighted sum of the previous gradients, which helps the optimizer to continue in the same direction if the gradient is consistent, reducing convergence time. The momentum term is also useful since even if a local minimum is encountered, if sufficient momentum is built up, then the descent will slow down, but still move out of the local minimum. Hence, the SGD method tends to work well with loss functions that may have local minima, but has an overall convex (bowl like) shape so that more often than not the gradients point to the global (or at least a decent) minimum.

I chose to use a fixed set of batches of images that I cycle through for each epoch, however before looping through the batches I shuffle the batches to introduce randomness into the gradient calculation. The stochastic nature of order of batches and hence the gradient calculation can help the optimization algorithm escape local minima, and can also help with generalization (reducing overfitting) since the network is not overfitting to the entire training set.

As a default value for the momentum parameter, I use 0.9, which is a common value for momentum. The learning rate is a hyperparameter that can be tuned to optimize convergence speed for a particular problem. Velocity is initially set to 0.

3.5 Genetic Algorithm

Because I am interested in exploring several evolutionary optimization strategies (outside of this project), and we already covered the details of evolutionary methods in class and homeworks, I chose to use the evosax package to save time (and not reinvent the wheel). Reimplementing the algorithms we did in class to leverage JAX gpu acceleration would take significant effort and provide little benefit since it has already been done in the evosax package. The evosax package contains several state of the art evolutionary optimization methods (and other global optimization methods), all implemented using JAX. It is important that for comparison of the methods that they all use the same underlying technology (JAX) to ensure a fair comparison.

SimpleGA, the GA I decided to use, is described in Such et al. (2017) [2]. This GA evaluates each individual in the population against the loss function (described prior). Between each generation, SimpleGA uses truncation selection to select the top individuals in the population (the 'elite') to be the parents of the next generation. Parents are chosen randomly from this elite population with replacement for crossover (this is not described as being part of the implementation in the paper, but the implementation in the evosax package does use crossover). After the children are created, noise (mutation) is added.

For the evolutionary optimization we have the following hyperparameters for SimpleGA:

- population_size: the number of individuals in the population
- cross_over_rate: probability of crossover occurring between genes (i.e. between weights and biases for two parent NN networks)
- elite_ratio: the fraction of highest fitness individuals from the population selected for reproduction
- sigma_init: the initial standard deviation of a normal distribution with zero mean used for adding 'noise' for the mutation of the weights and biases
- sigma_decay: the factor used for multiplicative decay of sigma_init after each generation
- sigma_limit: the minimum standard deviation of the mutation (after which it remains constant)
- init_min: the minimum value of the initial weights and biases (generated using a uniform distribution)
- init_max: the maximum value of the initial weights and biases (generated using a uniform distribution)

I chose the following hyperparameters for the evolutionary optimization as a starting point for experimentation:

Population size was limited by my gpu card memory. I used the max I could without problems, which is $\backsim$ 30. This is kept fixed for all the experiments.

cross_over_rate was set to 0.5, which is equivalent to a uniform cross-over. This is kept fixed for all the experiments (I did not explore different cross-over strategies).

elite_ratio was set to 0.25, so the top 25% of the population is selected for reproduction. Two individuals are randomly selected from this elite population to reproduce until the population of the next generation is filled (up to population_size). This will be a parameter that I will explore in the experiments.

sigma_init was initially set to 0.01, which is approximately equal to half the lowest standard deviation among the layer weights and biases for the NN trained with SGD (corresponds to the standard deviation of the biases for the first layer of the NN, see *Figure 8*). While the standard deviations for the weights and biases for the layers are not wildly different, I chose half the lowest because even this will result in a significant amount of added "noise" to the network parameters, resulting in significant exploration initially. An initial mutation of the same order of magnitude as the standard deviation of the NN parameters after optimization with SGDM is a reasonable starting point for exploration of the neural network landscape. This parameter will be explored in the experiments.

sigma_decay is the multiplicative decay rate that I calculate so that it results in a specified sigma_limit value (described next) after 2300 generations, at which point the optimization is terminated. This results in a factor of $\backsim$ 10 times more time to complete training relative to the mean convergence time using SGDM with optimal parameters. My thinking was that an order of magnitude more time is the most amount of increased computation that is reasonable to consider for the Evolutionary Optimization (EO) relative to the defacto standard SGDM optimization for NNs. If it takes more than an order of magnitude of time to train using EO, even if you find a better solution, it may not be worth (or practical) to use the EO method relative to SGDM.

sigma_limit was set to 0.001, which is $\backsim$ 10 times smaller than sigma_init. This parameter will be explored in the experiments.

init_max and init_min were set to +/- 0.3, which is a reasonable choice when looking at the statistics of the optimal weights and biases using SGDM discussed previously (see *Figure 8*). This is also the same range used for the initial weights and biases for the neural network optimization with SGDM, so comparison can be made on an equal footing. These parameters are kept fixed for all the experiments.

4 Results & Discussion

4.1 SGDM: Parameter Optimization

Before I can actually test the different training methods against each other, it is important that I find approximate optimal parameters to use for both methods so that they will perform on an equal playing field. Here, I use coordinate descent to approach the optimal parameters for the learning rate and momentum parameters in my SGDM algorithm.

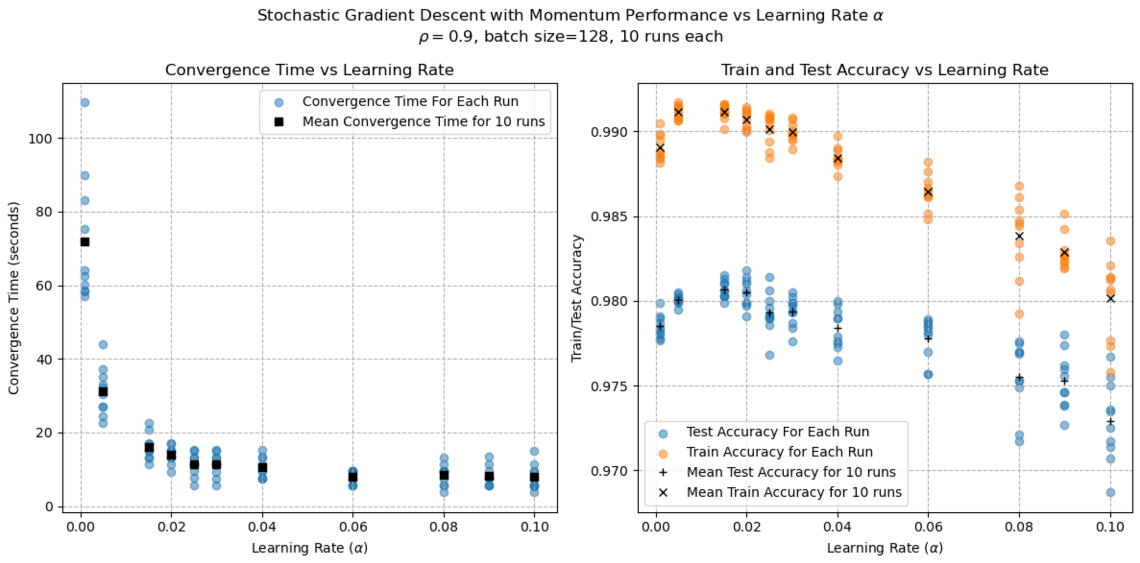


Figure 4. Convergence time of SGDM as a function of the learning rate to find what value of the learning rate produces the fastest convergence time.

As shown in the left plot in *Figure 4*, as the learning rate is increased the convergence time tends to decrease. However, if the learning rate is too high, the train and test accuracy suffers. Interestingly, if the learning rate is too low, the convergence time is not only slow, but the train and test accuracy also suffers (perhaps due to the greater chance of getting caught in a local minimum with a small learning rate). Based on the above results, 0.02 seems to be a good choice for the learning rate with good convergence speed and consistent convergence to network parameters with high train/test accuracy. I now fix the learning rate at 0.02 and explore the effect of different values for the momentum parameter ρ.

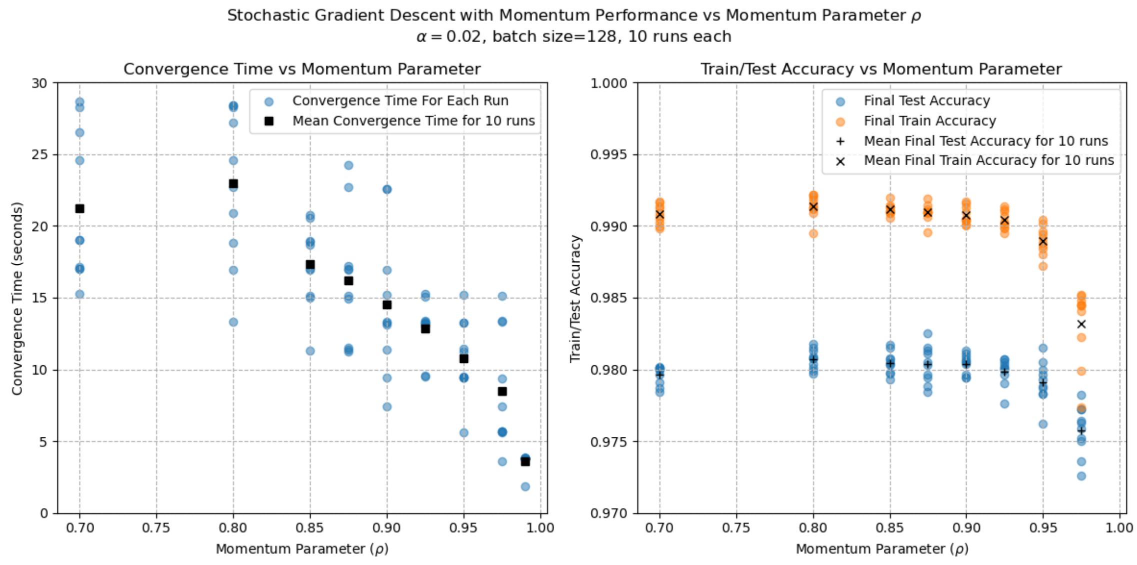


Figure 5. Convergence time of SGDM as a function of the momentum parameter to find what value of the momentum parameter produces the fastest convergence time.

As shown in the left plot in *Figure 5*, higher momentum parameter values result in faster convergence. However, as shown in the right plot, if the momentum value is too high the network parameters converge to a solution with lower test and train accuracy. For the NN and training data being considered here, 0.9 is a very good choice for the momentum parameter that results in the fastest convergence without compromising accuracy. Because in the prior experiment the learning rate of 0.02 was found to be an 'optimal' choice with a momentum parameter of 0.9, and in above experiment the momentum parameter of 0.9 was found to be an 'optimal' choice with a learning rate of 0.02, we can say that the pair of learning rate and momentum parameter of 0.02 and 0.9, respectively, is an 'optimal' pair for training the NN with SGD with momentum. This would not necessarily be the case if the optimal momentum parameter was different from the 0.9 assumed when exploring the learning rate, in which case I would have needed to iterate between the two hyperparameters to find the optimal pair (which I had to do, and am only presenting final results here).

4.2 SGDM: Training Consistency

Now that I have determined the 'optimal' learning rate and momentum parameters, I train the network with 100 different random initializations to get insight into the variability of the training process.

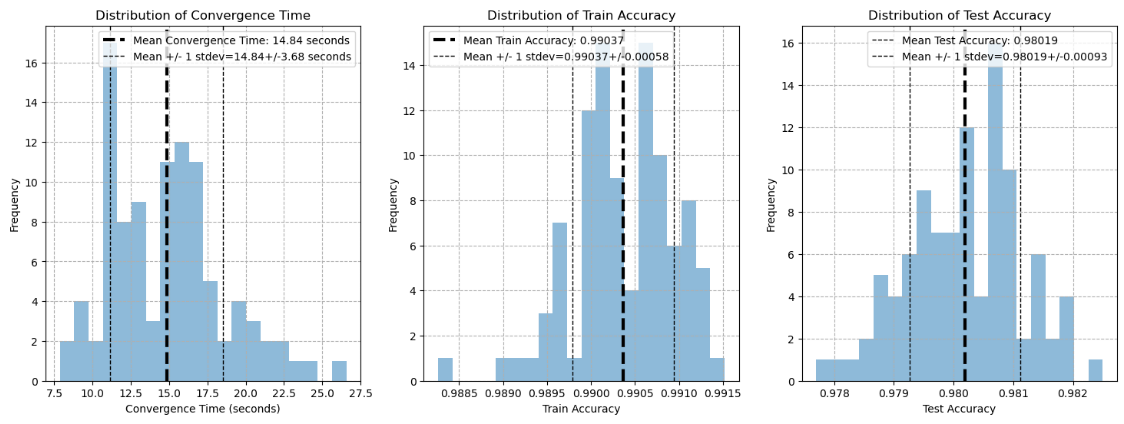


Figure 6. Distribution of convergence time, training set accuracy, and testing set accuracy after network training using SGDM for 100 random initializations of the network.

As shown in *Figure 6*, both the train and test accuracy have very low standard deviations, indicating that the training process is very consistent. This may indicate that perhaps the loss function being used, even though highly nonlinear given the nonlinear activation functions, may not have many (or perhaps any) local minima, or if it does, than the various local minima may all have very similar loss. The Train accuracy mean is 99.04%, and the test accuracy mean is slightly lower at 98.02%. The test accuracy is slightly lower than the train accuracy, which is expected since the network is optimized for the training data. The test accuracy is consistently high however, which indicates that the network is generalizing well to the test data.

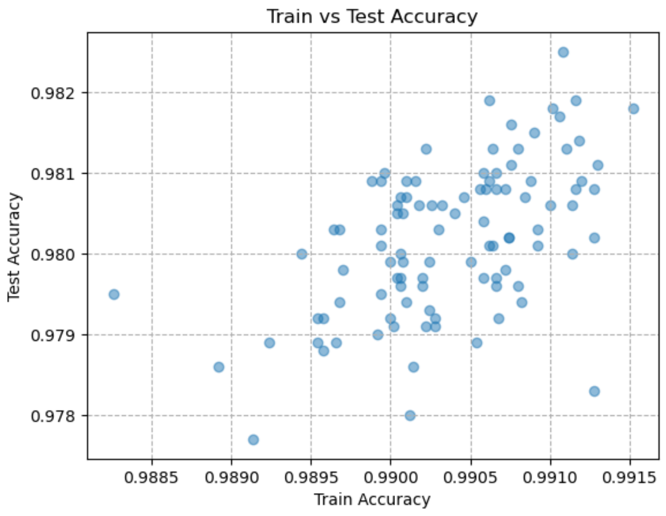


Figure 7. Test accuracy plotted against training accuracy after training using SGDM for 100 random initializations of the network.

Next I plot the test accuracy versus the train accuracy for each of the 100 random initializations. If the training accuracy tends to be high when the test accuracy is low, then it is an indication that the NN is overfitting to the training data. However, as shown in *Figure 7* this is not the case indicating that we are not over fitting.

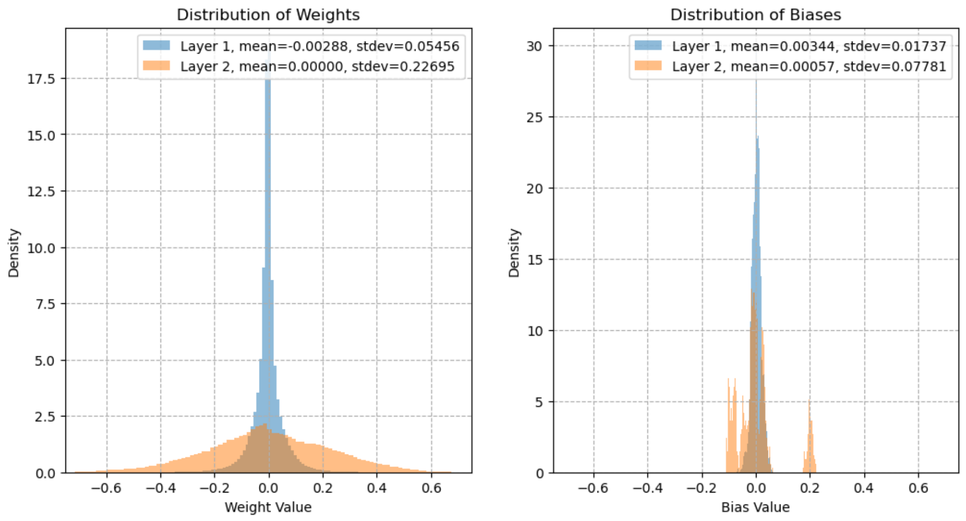


Figure 8. Distribution of weight and bias values across differing layers in the neural network.

As shown in *Figure 8*, weights and biases don't seem to be wildly different, nor is there a huge difference from one layer to the next. The reason I check this is that the network weights and biases for all the layers are initialized using the same uniform distribution, so it makes sense to check that the weights and biases for each layer are not wildly different for the ensemble of optimized networks since otherwise a different initialization strategy would be appropriate between the weights and biases and each layer.

4.3 GA: Parameter Optimization

Next, I repeat the same process of parameter optimization using coordinate descent for the SimpleGA algorithm on the sigma_limit and sigma_init parameters.

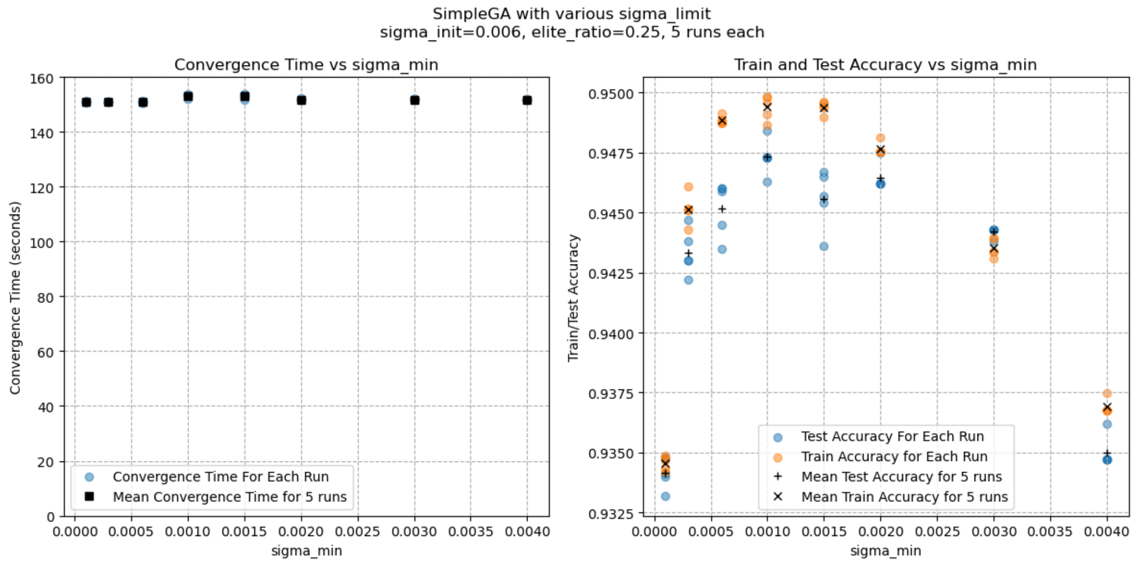


Figure 9. Convergence time of SimpleGA as a function of the sigma_limit parameter to find what value of sigma_limit produces the fastest convergence time.

As shown in *Figure 9*, the optimal sigma_limit is 0.001. Since this is the standard deviation of the normally distributed mutation 'noise', it makes sense that it is small at the end of the optimization since the network parameters are close to the optimal values and we don't want to disrupt them too much so as not to converge.

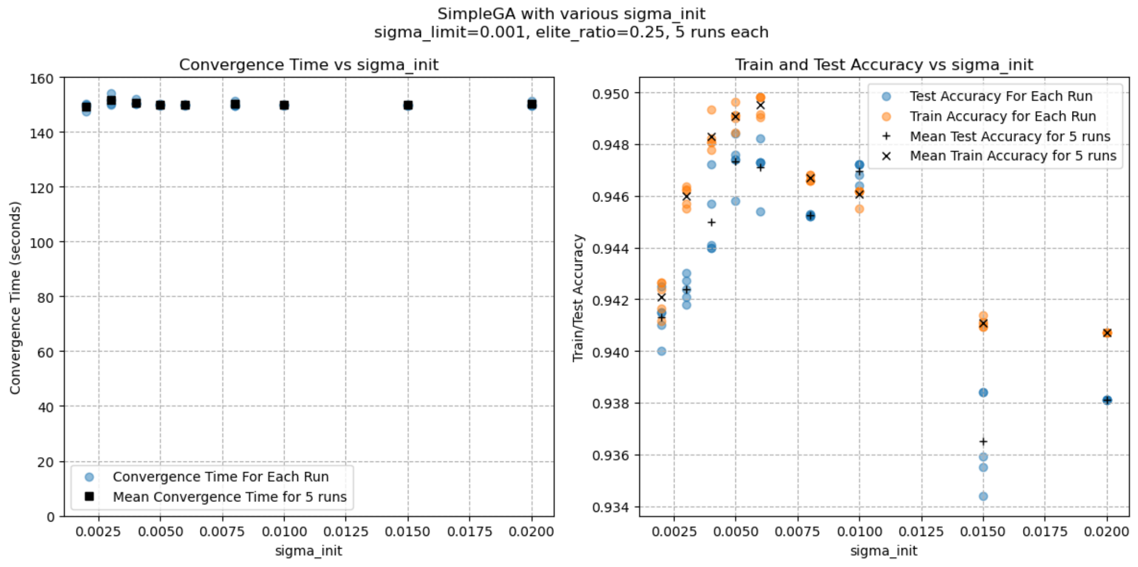


Figure 10. Convergence time of SimpleGA as a function of the sigma_init parameter to find what value of sigma_init produces the fastest convergence time.

As shown in *Figure 10*, the optimal sigma_init is 0.006.

4.4 SGDM versus GA

Finally, with optimal parameters found, I can run each algorithm and compare their performances against each other.

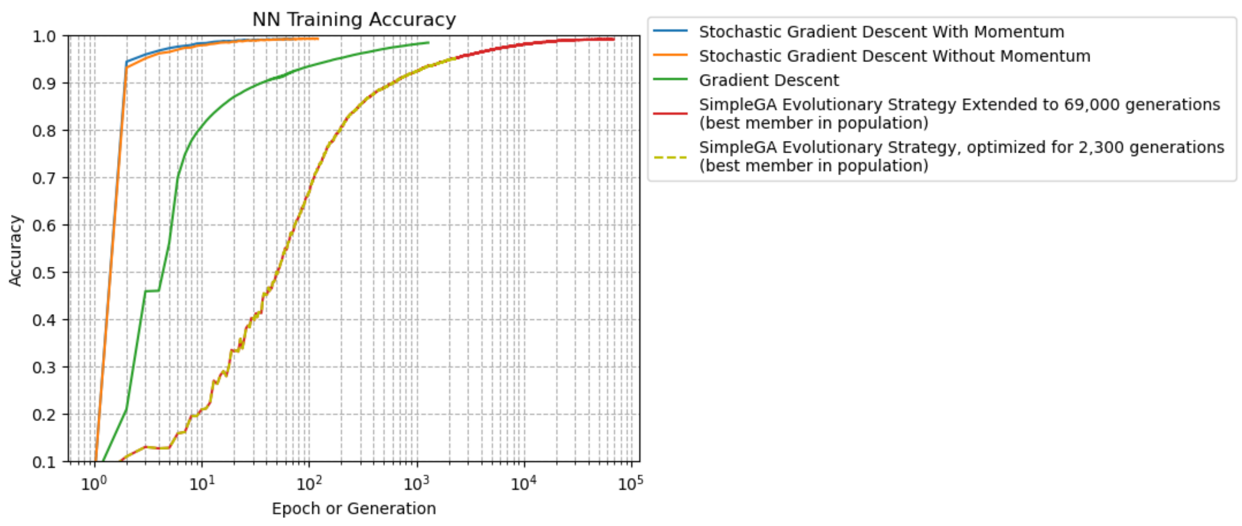
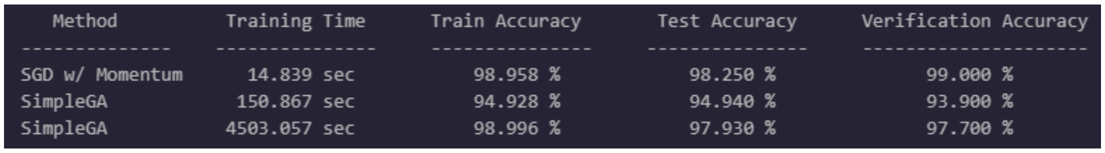


Figure 11. Plot of accuracy results over each generation for training, testing, and verification datasets for GD, SGD, SGDM, SimpleGA for 2,300 generations, and SimpleGA for 69,000 generations. All methods use optimized hyperparameters.



```
   Method          Training Time     Train Accuracy      Test Accuracy      Verification Accuracy
-------------     ---------------   ---------------    ---------------    ---------------------
SGD w/ Momentum     14.839 sec         98.958 %           98.250 %             99.000 %
SimpleGA           150.867 sec         94.928 %           94.940 %             93.900 %
SimpleGA          4503.057 sec         98.996 %           97.930 %             97.700 %
```

Figure 12. Table of train time and accuracy results for training, testing, and verification datasets for SGDM, SimpleGA for 2,300 generations, and SimpleGA for 69,000 generations. All methods use optimized hyperparameters.

As shown in *Figure 11*, I run gradient descent (GD), stochastic gradient descent (SGD), stochastic gradient descent with momentum (SGDM), and SimpleGA all with optimal parameters shown, but I also include an extra long simulation of SimpleGA that runs for tens of thousands of generations. While I originally wanted to explore the performance of the GA in what I considered to be a 'reasonable' time when compared to SGDM (in the first case, about 10 times the run time), I also wanted to see what the results would look like if I let the GA run for far longer.

SimpleGA, even when given orders of magnitude more time to converge on local minima, is only able to achieve similar network performance on the training set when compared to SGDM. Contrarily, it performs noticeably worse than SGDM for the verification set, and slightly worse for the test set. Perhaps SimpleGA is more prone to overfitting.

This again raises the point discussed at the beginning of this project, which is that perhaps local minima with good performance are not very difficult to find using this loss function. While SGDM performs well in the case explored here, this cannot be used to reach the conclusion that the SGDM algorithm is simply better at training neural networks in general. The relative performances of both methods could change dramatically depending on the loss function used and the ruggedness of the landscape being optimized.

5 Conclusion & Future Work

In my final results, SGDM outperforms SimpleGA both in terms of accuracy and convergence time. However, it is important to recognize that this is specific only to the MNIST dataset that I used to train the network. For fitness landscapes that are less smooth (more rugged) and have many more local optima, it is possible that the GA would be competitive as described in the paper [5] that motivated this project. In those cases, it can be possible for the GA to be competitive with or outperform SGDM even if convergence takes longer.

For this reason, the main takeaway from this project is that the method by which we optimize for our loss function is contingent on the type of problem that we are trying to optimize for. In this case, I found that SGDM was a better performer.

It's also important to note that for some loss functions, SGDM may not even be an option at all due to lack of differentiability. If this is the case, a GA might not just perform better, but be necessary.

Future work here would involve exploring other loss functions. For some loss functions, Such et al. [5] found improved performance using the GA when training a neural network to play some (but not all) Atari 2600 games. Going forward, it would be valuable to explore what actually distinguishes a better performing GA from a better performing GD algorithm. How *does* the loss function affect this?

6 References

[1] Ahmad, F., Isa, N. A., Osman, M. K., & Hussain, Z. (2010). Performance comparison of gradient descent and genetic algorithm based artificial neural networks training. 2010 10th International Conference on Intelligent Systems Design and Applications, 604-609. https://doi.org/10.1109/isda.2010.5687199

[2] Goodfellow, I., Bengio, Y. & Courville, A. (2016). Deep Learning. MIT Press. 181.

[3] Morse, G., & Stanley, K. O. (2016). Simple evolutionary optimization can rival stochastic gradient descent in Neural Networks. Proceedings of the Genetic and Evolutionary Computation Conference 2016. https://doi.org/10.1145/2908812.2908916

[4] Pendharkar, P. C. (2007). A comparison of gradient ascent, gradient descent and genetic-algorithm-based artificial neural networks for the binary classification problem. Expert Systems, 24(2), 65-86. https://doi.org/10.1111/j.1468-0394.2007.00421.x

[5] Such, F. P. et al. (2017). Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. CoRR abs/1712.06567. https://doi.org/10.48550/arXiv.1712.06567

[6] Xue, Y., Tong, Y. & Neri, F. (2023). A hybrid training algorithm based on gradient descent and evolutionary computation. Appl Intell 53, 21465-21482. https://doi.org/10.1007/s10489-023-04595-4