Project Report: “Patrick, A new Optimizer”

Pranav Pishawikar pmp15101

**1.0 Introduction and Problem Statement**

An Optimizer is absolutely vital to Machine Learning. During the training process, tweaks and changes need to be made to the parameters (weights) of our model to try and minimize the loss in our model in order to make our predictions as accurate as possible.

This is where the function of the optimizer comes in. An optimizer, in essence, tries to make the model more accurate by changing these parameters. The loss function essentially acts as a guide to the model, telling the optimizer when it is moving in the right or wrong direction.

Put simply, an optimizer in general takes in an input function and a starting point and spits out a direction to travel to reduce the function. A good Optimizer can get us to the function’s optimum accurately and with minimal function evaluations. A brute force method would be to evaluate the function at all points and thus find the minimum. However, this is computationally extremely expensive as the range of the function is usually fairly large (if not open) and in complex machine learning model, these evaluations can take up quite some time.

The foundations for optimizers are based in Mathematics. Fermat and Legrange founded the basics of calculus based Optimization in the early 1700s. Newton and Gauss were the first to propose iterative methods, again in the early 1700s. George B. Dantzig and Leonid Kantorovich were responsible for Linear Programming in the early 1900s. Of note here is that programming in this context does not refer to computer programming, but comes from the use of program by the United States military to refer to proposed training and logistics schedules, which were the problems Dantzig studied at that time. Dangzit later published the Simplex Algorithm in 1947. However, the main methods involved in optimization for machine learning have been largely the same as the ones that were first proposed in the 1700s.

The objective of my project is to come up with a new optimizer based on adam that should in certain cases be able to provide more accurate and faster results.

Importance and application of the problem

As mentioned earlier, a good optimizer should be computationally inexpensive, should require a minimum number of function evaluations (or iterations) in order to reach a minimum and should be accurate.

Having a good optimizer should in theory improve the speed of the machine learning process while also making the model predictions as accurate as possible.

**2.0 A Survey of Known Algorithms**

Although there are many optimizers that may be used for machine learning, the most commonly used ones include gradient descent, stochastic gradient descent, mini-batch gradient descent, adam, momentum, RMSprop and adagrad. For the sake of brevity, in this section, we will only discuss two of these techniques. Stochastic gradient descent will be briefly described and adam will be described in a bit more detail. We will also discuss simulated annealing, which is a probabilistic technique that is not commonly used for machine learning as this will be important to set up the foundation for discussions in the future sections. To this effect, we will also have a brief discussion of the effect of learning rates on model training in this section.

*2.1 Stochastic Gradient Descent*

Stochastic gradient descent (hereby abbreviated SGD) is an iterative method for optimizing an objective function. Of note here is that it is assumed that the function in consideration has the appropriate smoothness properties. Smoothness in this context means that the function must be continuous as well as differentiable.

This is important because the root of the optimization technique relies on computing the gradient for the function being evaluated. The gradient is defined as the first order partial derivative of the function with respect to each of its parameters. It can be mathematically proven that this gradient evaluation actually gives us the direction in which to travel in order to reduce the function optimally. The proof for this is out of the scope of this report and will be left to references [2].

SGD can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data).

*2.2 Adam Optimizer*

The adam optimizer is probably the most used optimizer for deep neural networks. This is due to several important advantages that the algorithm has. It is fairly straightforward to implement, computationally pretty efficient, has little memory requirements and is overall very well suited for problems that are large in terms of data and/or parameters.

The root of the optimization technique, like SGD lies in computing the gradients of the function. However, adam does a few things differently in order to provide major advantages over SGD. The pseudo code for adam algorithm is provided below followed by a description of what happens at each stage of the algorithm.

1 **while** θt not converged **do:**

2

3 t ← t + 1

4 gt ← ∇Ө ft(θt-1) (Get gradients w.r.t. stochastic

5 objective at time step t)

6 mt ← β1 · mt-1 + (1 − β1) · gt (Update biased 1st

7 moment estimate)

8 vt ← β2 · vt-1 + (1 − β2) · gt2 (Update biased 2nd

9 raw moment estimate)

10 mt ← mt/(1 − β1t ) (Compute bias-corrected 1st

11 moment estimate)

12 vt← vt/(1 − β2t ) (Compute bias-corrected 2nd raw 13 moment estimate)

14

15 θt ← θt-1  − α·mt/( √ vt + ε ) (Update parameters)

16

17 **end while**

18

19 **return** θt (Resulting parameters)

The adam optimizer takes two additional inputs than SGD, ­. These parameters may in theory be changed during execution to any values in the range [0, 1). However, default values are provided if none are specified.

Lines 1 to 17 consists of the iterative loop the calculated values for which are returned in line 19. At the start of each execution of the loop, the time step, is incremented by 1 (line 3).

The stochastic gradient, ­ is calculated in lines 4 and 5. However, unlike SGD, these values are not directly used. The updated biased first moment, estimate is calculated in lines 6, 7. This is where adam deviates from SGD. In essence, the updated moment comes in part (­) from the moment calculated in the previous time step, and the remainder (­) from the current gradient calculated in lines 4 and 5. This feature serves to give adam a better idea of the general direction in which the function is headed rather than just the instantaneous direction as with SGD.

Lines 8 and 9 calculate the variance, defined as the square (dot product with self) of the gradient. Just as the gradient gives the direction to travel, it can be proven mathematically (again outside the scope of this report) that the square of the gradient gives the variance in the gradients over time. Again, just like with the gradient itself, the variance also uses information from the previous iterations in order to get a better estimate of the average variance of the function instead of just the instantaneous variance, this time, using the factor ­.

Lines 10 to 14 essentially work to perform bias correction for the momentum and variance. In practice, these are not always implemented. However, in theory, they help to reduce the dependence of the algorithm on the initial values of moment and variance during the start of the learning period and also at plateau points on the function being evaluated.

Finally, in line 15, the parameters are re evaluated using the step size, and the ratio of the moment to the square root of the variance (which can be thought of as the signal to noise ration with some abuse of notation). The factor is used to prevent issues due to vanishing gradients where we never want to divide by zero.

*2.3 Simulated Annealing*

Simulated Annealing is a probabilistic technique that can be used for function optimization. It is used to prioritize finding the global optimum of a given objective function over a larger number of functional evaluations rather than to find a local optimum quickly. Due to the fact that the functions in question are fairly irregular in nature, there is no guarantee at the end of an optimization that the optimizer has arrived at the global optimum rather than a local one. Although simulated annealing does not guarantee this either, it accepts only the best local optimum with high probability.

At the root of Simulated Annealing is the temperature function. In essence, this temperature is a representation of the probability that the optimizer will move to a point in the function that has a worse solution than the previous one. Over time, the value of this temperature is reduced, giving us only better results after the temperature reaches zero.

By accepting worse solutions at the start, the optimizer essentially ends up exploring a larger range of the function and tends to settle as the temperature reduces, in the portion of the function with the deepest valleys. This technique is often used when finding the approximate global optimum is more important than finding an accurate local optimum.

*2.4 Effect of Learning Rate on Model Training*

Although a detailed explanation on this topic is outside of the scope of this report, intuitively, a step size that is too large will initially reduce the loss in the model very quickly. However, over time, it will never be able to achieve the accuracy that you would be able to achieve with a finer step size. On the flipside, a learning rate that is too small will be very slow in training the model accurately and even after multiple evaluations, will not be able to get to the optimum as shown in the figure below. It is thus extremely important to choose a learning rate that is not too large, but not too small either. Of note is that learning rates cannot be determined analytically. They usually involve trial and error in order to figure out the best learning rates. For a more detailed explanation on this topic, we refer you to [6].

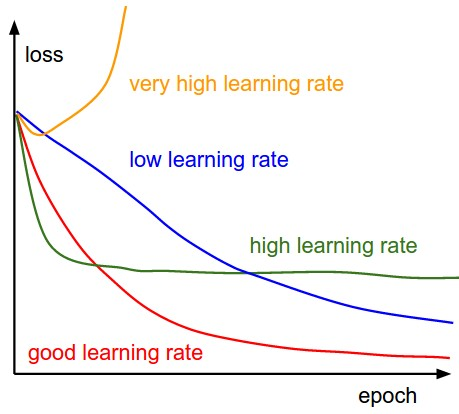


figure 1: effect of learning rate on loss during training

**3.0 Patrick, a New Optimization Algorithm**

As mentioned earlier, the Patrick algorithm is based on adam. Two separate algorithms were made and implemented. The logic and reasoning of each implementation are discussed after providing the pseudo code for each.

*3.1 Algorithm 1*

1 **while** θt not converged **do:**

2

3 t ← t + 1

4 gt ← ∇Ө ft(θt-1) (Get gradients w.r.t. stochastic

5 objective at time step t)

6 mt ← β1 · mt-1 + (1 − β1) · gt (Update biased 1st

7 moment estimate)

8 vt ← β2 · vt-1 + (1 − β2) · gt2 (Update biased 2nd

9 raw moment estimate)

10 mt ← mt/(1 − β1t ) (Compute bias-corrected 1st

11 moment estimate)

12 vt← vt/(1 − β2t ) (Compute bias-corrected 2nd raw 13 moment estimate)

14 Tt← (Tt)exp (Update the temperature)

15

16 θt ← θt-1  − α·Tt·mt/( √ vt + ε ) (Update

17 parameters)

18

19 **end while**

20

21 **return** θt (Resulting parameters)

As mentioned earlier, the root of this algorithm comes from adam. The only real difference is it requires two additional parameters, the temperature, t and the exponent exp, which may be provided during execution or will get default values if not provided. T is in the range [1, ) while exp is in the range [0, 1). In line 16, the algorithm effectively multiplies the step size by the current temperature. As the number of iterations increases, due to the exp being smaller than one, the current temperature decays and tends to 1. Thus we effectively start of with a larger learning rate but ultimately end with a good learning rate as the number of iterations increase.

This in effect leverages the benefit of higher learning rates mentioned in the previous section without compromising on the accuracy of the results during the later learning phase.

*3.2 Algorithm 2*

1 **while** θt not converged **do:**

2

3 t ← t + 1

4 gt ← ∇Ө ft(θt-1) (Get gradients w.r.t. stochastic

5 objective at time step t)

6 mt ← β1 · mt-1 + (1 − β1) · gt (Update biased 1st

7 moment estimate)

8 vt ← β2 · vt-1 + (1 − β2) · gt2 (Update biased 2nd

9 raw moment estimate)

10 mt ← mt/(1 − β1t ) (Compute bias-corrected 1st

11 moment estimate)

12 vt← vt/(1 − β2t ) (Compute bias-corrected 2nd raw 13 moment estimate)

14 Tt← (Tt)exp (Update the temperature)

15

16 **if** Tt-1 > ε**:**

17 R=randrange[1,r]

18 **if** Tt < R:

19 R=1

20

21 **else:**

22 R=1

23

24 θt ← θt-1  − α·R·mt/( √ vt + ε ) (Update

25 parameters)

26

27 **end while**

28

29 **return** θt (Resulting parameters)

Algorithm 2 is very similar to algorithm 1. The only difference is it uses a different multiplicative factor, R in the final function evaluation instead of using the temperature directly. To do this, the algorithm also takes in an additional input ‘range parameter’, r. This again may be specified during execution or the default provided value could be used.

In algorithm 2 we try to leverage the benefits of Simulated Annealing mentioned earlier. Until the temperature reaches 1, we may accept randomly generated values for the multiplicative factor, R in the range [1 to r]. Once the temperature approaches 1 (within a factor of ), the multiplicative factor is always R=1 (lines 21 and 22).

Due to the way the algorithm is written, the probability that we accept a different value for R other than 1 reduces as the temperature reduces (lines 18 and 19). The largest accepted values of R also reduce as the number of time steps increase and temperature reduces due to the same loop.

*3.3 Implementation*

The implementation for this algorithm was done in the python 3.7 tensorflow framework. Many dependencies need to be taken care of and some of the core tensorflow files including training.py, \_\_init.py\_\_, gen\_training\_ops.py, etc. within the framework/training and core/training directories need to be modified in order to accommodate an additional optimizer in the tensorflow library. Multiple lines need to be modified within these files and therefore, the implementation details will be left out of this report. Instead a google docs link [7] is available at the end of the paper in the additional resources and references section. This should provide a detailed report on what changes need to made to accommodate Patrick on a python 3.7 tensorflow environment. The link to the optimizer algorithms is also provided in [8]. This file needs to be pasted in the /python/training folder.

*3.4 Testing and Results*

There were two distinct kinds of testing that I was hoping to do with each of the two algorithms mentioned before. First, the initial testing that would help determine the default values for the parameters ‘T’, ‘exp’, and ‘r’ that should be used in order to get the best possible optimizer. Second, a test using different optimizers to train the same neural networks in order to get a good comparison of the performance of the Patrick optimizer against others like adam, adagrad and SGD.

Unfortunately, just building and implementing the optimizer took a substantial amount of time, leaving me with just a week for testing.

Due to this, and the fact that training a neural network takes a large amount of time, I was not able to complete all the tests I had in mind. I did complete some tests to help determine good (but not optimal) values for the parameters ‘T’ and ‘exp’ for Algorithm 1. I was unable to find good values for ‘r’ for algorithm 2 in my testing, although given more time, I am sure I can get them. On account of the fact that training Neural Networks takes time, I was also not able to generate many results for comparing the performance of Algorithm 1 against other Optimizers. However, what follows are details of the network architecture I used for testing and the results that I did manage to get.

I used a CNN with three convolutional layers (5x5, 3x3, 3x3) and four fully connected layers (1500, 500, 500, 10) with dropout and batch normalization to train using the CIFAR10 database. All my tests were done using batch sizes of 100 and just 30000 iterations. Admittedly, this is not at all ideal, but due to the time constraints, I had to be able to do many tests quickly and therefore made a network that would provide about 40 to 50 % accuracy in 4 to 5 hours of learning.

After some initial testing I found that the values of ‘T’= 10 and ‘exp’= 100/101 seem to work fairly well for Algorithm 1. I was not able to find good values for ‘r’ for Algorithm 2. The results for the latest image classification test using Algorithm 1 (Red) and Adam (Green) are displayed below

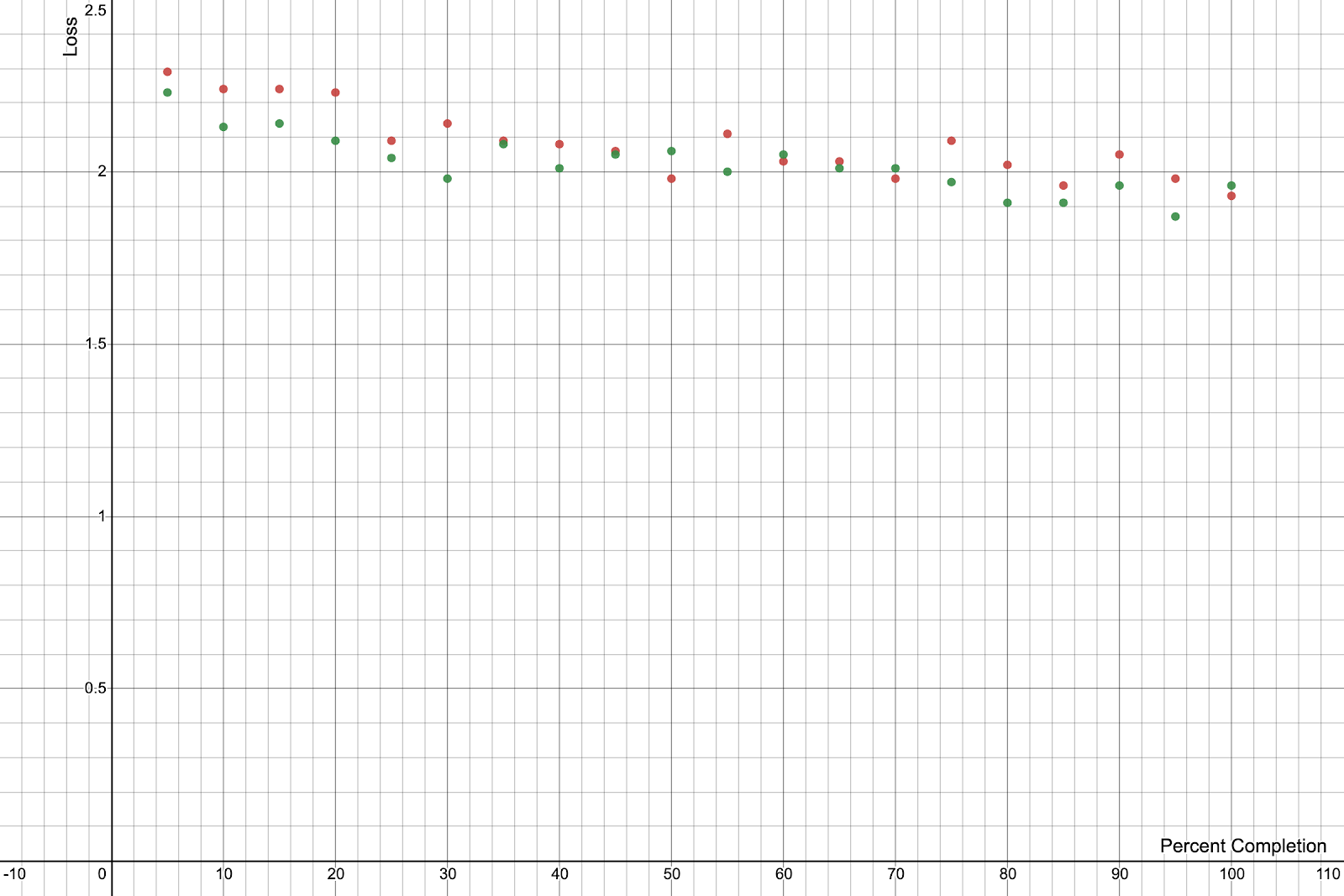


figure 2 : Loss vs Percent Completion (Adam = Green, Patrick = Red)

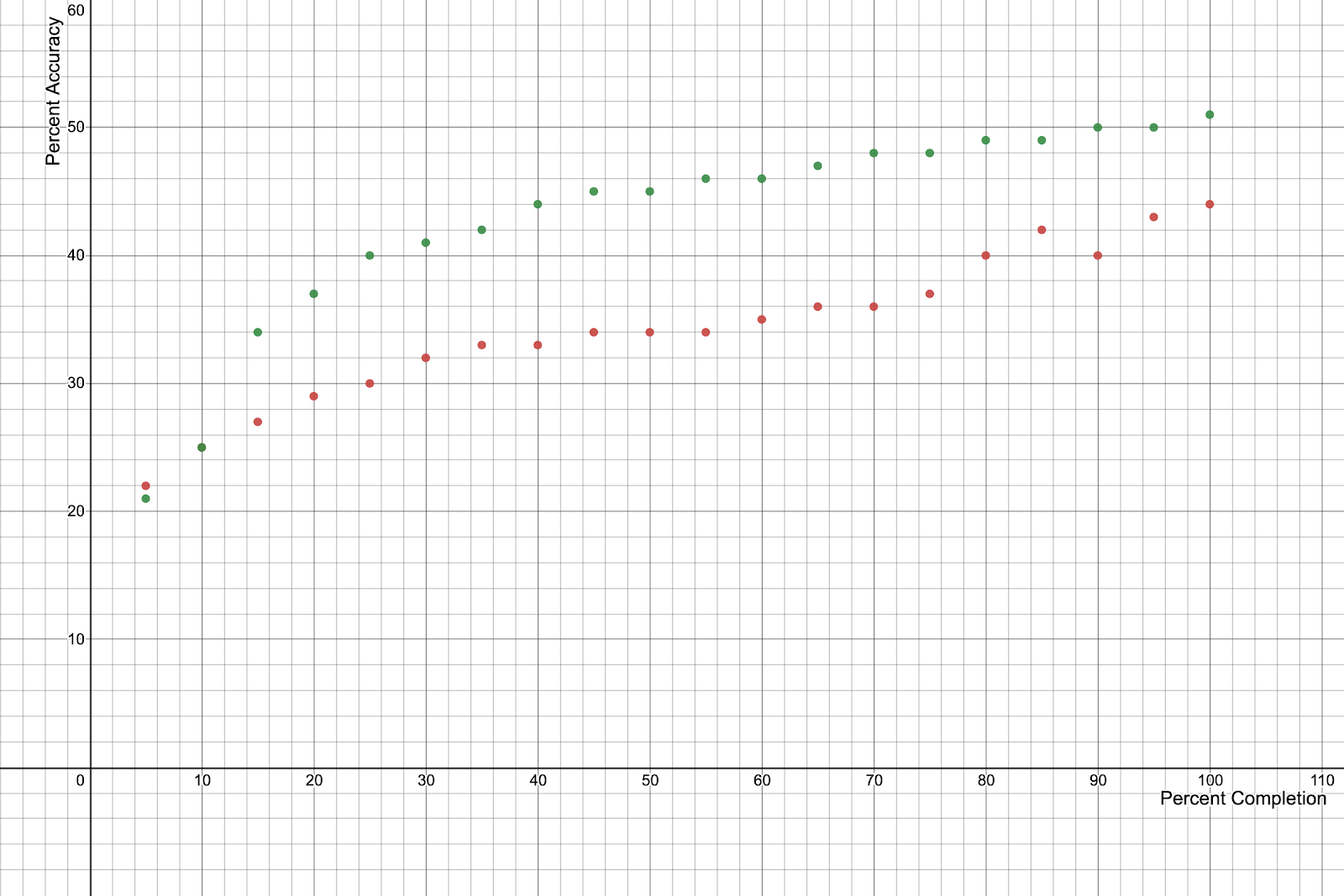


figure 3 : Accuracy vs Percent Completion (Adam = Green, Patrick = Red)

**4.0 Conclusions**

Obviously the results provided above are not at all conclusive. These tests were very short and are just a single run case. To get a better idea of how the optimizers compare, much longer tests are needed. During my testing with various parameter values for ‘T’ and ‘exp’, I had multiple runs where I got a final accuracy of more than the 51% accuracy that the single run portrayed above using adam achieved. However, I did not record these readings as I was still in the first phase of testing. It is safe to say that Patrick using Algorithm 1 performs just as good as if not better than adam for the particular network architecture that I used.

There is tremendous scope for future work with this project. First off, an extensive amount of testing needs to be done in order to determine the optimal parameter values for both algorithm 1 and 2. This involves testing for longer periods, using different parameter values on different problems (not just image recognition) using different network architectures.

Tuning these parameters will no doubt improve the performance of this algorithm. More work also needs to be done in improving the implementation itself. At the moment, the current implementation of the optimizer does not support parallel computing like the other optimizers for implemented in tensorflow do.

Finally, it is also necessary to test the optimizer once the optimal parameters have been found against other optimizers like adam, SGD, and RMSprop for different types of problems on different types of networks.

**5.0 References and additional resources**

[1] <https://algorithmia.com/blog/introduction-to-optimizers>

[2] ‘Large-Scale Machine Learning with Stochastic Gradient Descent’ L´eon Bottou

[3] <https://en.wikipedia.org/wiki/Stochastic_gradient_descent>

[4] ‘ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION’ Diederik P. Kingma, Jimmy Lei Ba

[5] <https://en.wikipedia.org/wiki/Simulated_annealing>

[6] <https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10>

[7] <https://docs.google.com/document/d/1-xRAupaU-8TgIjc-hC2M5THb7hymwBqYnTK6V-HjeU4/edit?usp=sharing>

[8] <https://drive.google.com/file/d/1IDsxOKtFRokt-MVqafRcr9pKNWd9VTe3/view?usp=sharing>