# Optimising Neural Networks via Quasi-Newton, Newton Conjugate Gradient, Newton Conjugate Gradient Trust-Region Methods

Jayson Paul Salkey
ID: 17045589
Numerical Optimisation

April 7, 2018

**Abstract**

In the machine learning field, the effective training of neural networks through back-propagation in tandem with first order methods has remained the de-facto standard in effectively learning tasks. First order methods have remained in favour due to their low computational cost in terms of time and space complexities of computing gradients when compared to higher-order methods. However, when compared to higher order methods, the convergence rate to solutions is much slower. In fact, when considering deep neural networks of increasing large number of layers and units, training becomes an extremely slow process resulting in under-fitting on a training set of data. In this study, we study the effect of quasi-newton and conjugate gradient optimisation techniques on training deep neural networks of various architectures.

## 1 Data and Framework

Seeing that implementing a generic neural network framework architecture that can easily be extended with line search and trust region methods is outside of the scope of Tensorflow's computational graph ability, I have decided to write my own generic deep learning framework from scratch and have included this code in the ipython notebook. It was created with the idea of easily adding custom optimisers with line search and trust region methods in mind. I have used this framework for the case studies that I investigated in this report.

In order to conduct simulations on our methods, we have to decided to utilise the classical experiment framework of classifying handwritten digits from a large image dataset, otherwise known as the MNIST[LeC] dataset. The MNIST dataset is a database of 70,000 handwritten digits that have been size-normalised and centered in their respective images. Traditionally, this database is split into 2 sections, a 60,000 image training set, and a 10,000 image testing set. Each image in the dataset is represented by a 28 x 28 binary grid of pixels. Each image is also associated with a single digit in the base-10 number system. This digit is the image's classification or the labeled dependent variable of the image's pixels.

Figure 1: Samples from MNIST.[mni]

# 2 Problem

## 2.1 Loss Function

The goal of our task is to generate a hypothesis function from a training set of data, $T_D = \{(\mathbf{x_i}, \mathbf{y_i})\}^m$, $\mathbf{x_i}$, $\mathbf{y_i} \in \mathbb{R}^n$ that effectively maps from $\mathbf{x_i} \rightarrow \mathbf{y_i}$. In order to effectively track the performance of our model as we process samples from our training dataset, we must introduce a loss. In figure 2, we show the loss of a neural network as it trains over entire epochs of data utilising standard gradient descent.

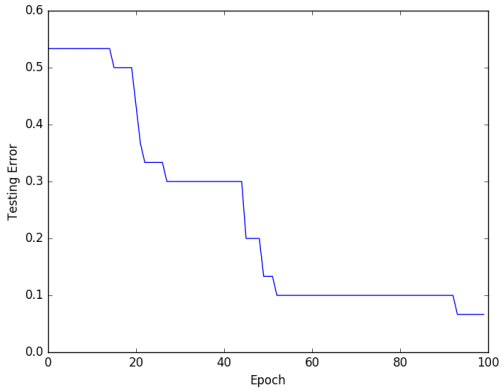


Figure 2: Neural network loss over a number of epochs of data.

In multi-label classification cases, like MNIST, we seek to define the loss in terms of the probabilities of each classification label. A well used loss function in this task is the cross-entropy. Let $q(a)$ be the output of the neural network and the estimate of a true probability distribution, $p(a)$. Also let $\langle p,\ q \rangle_{KL}$ be the Kullback-Leibler divergence. Finally, let $\theta$ be the parameters of

the neural network. In this case, the discrete variable a represents the 10 possible classes of the MNIST dataset. We can define the cross entropy loss on a single training example as,

$$L = -\sum_{a} p(a) log(q(a))$$

when considering discrete variables,
or more succintly, in terms of the original notation of the training set,

$$L(\theta) = \langle -\mathbf{y},\ log(\widehat{\mathbf{y}}) \rangle$$

where, $\widehat{\mathbf{y}}$ is your network's output vector and we are computing the inner product. Often one will see this as an empirical average over all examples in a batch-setting or entire epoch as,

$$L(\theta) = \frac{1}{m} \sum \langle -\mathbf{y_i},\ log(\widehat{\mathbf{y}}_\mathbf{i}) \rangle$$

It is this loss function that we are going to minimise with more advanced second-order approximations.

### 2.2 Assumptions

We will assume some standard technical assumptions for providing a stage for this problem, We will assume that $L(\theta)$ has Lipschitz continuous gradients

$$||\nabla L(\theta) - \nabla L(\theta')|| \leq C||\theta - \theta'||$$

We will assume that $L(\theta)$ is strongly convex, it is possible that this is only near some local minimum however.

$$L(\theta + h) \geq L(\theta) + \nabla L(\theta)' h + \frac{1}{2} h' \nabla^2 L(\theta) h$$

We will also assume that gradients and higher order quantities are computed without stochasticity.

## 3 Optimisation Methods

### 3.1 Line Search

Typically in training deep networks, the learning rate of the optimisers is usually fixed to lay between, $\alpha \in (0, 1]$. This is chosen as it is usually very computationally expensive to perform a line search minimisation in the direction of the computed search direction. However, in our experiments, to showcase the difference in learning these parameters of the network across different optimisers, network depths, and mini-batch sizes, we utilise the simple algorithm, backtracking line search, in order to enforce a sufficient decrease in the loss function by requiring that we adhere to the Armijo condition, $f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k' p_k$.

**Algorithm 1:** Backtracking Line Search

**Result:** $\alpha$
$\alpha_0 \leftarrow 1.$;
$\rho \in (0, 1)$;
$c \in (0, 1)$;
$\alpha \leftarrow \alpha_0$;
**while** $L(\theta_k + \alpha p_k) > L(\theta_k) + c\alpha \nabla L(\theta_k)' p_k$ **do**
  $\alpha = \rho\alpha$ ;
**end**

## 3.2 Trust Region

For Trust region methods, we cannot use line search as we now have a region in which we trust or do not trust our model.

**Algorithm 2:** Trust Region

**Result:**
**while** *not converged* **do**
  $p_k \leftarrow solver(\cdot)$;
  $m_0 \leftarrow L(\theta_k)$;
  $m_{p_k} \leftarrow L(\theta_k) + \nabla L(\theta_k)' p_k + \frac{1}{2} p_k' \nabla^2 L(\theta_k) p_k$;
  $\rho \leftarrow \frac{L(\theta_k) - L(\theta_k + p_k)}{m_{p_k} - m_0}$;
  **if** $\rho < .25$ **then**
    $\Delta = .25 * \Delta$;
  **else**
    **if** $\rho > .75 and norm(p_k) = \Delta$ **then**
      $\Delta = min(2 * \Delta, 1)$;
    **end**
  **end**
  **if** $\rho > \eta$ **then**
    $\theta_k = \theta_k + p_k$;
  **end**
**end**

For the Hessian computation, we make use of a finite differences approach where such that we can attain a Hessian vector product.

$$\nabla^2 f_k d \approx \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h},\ h \approx 10e - 8$$

## 3.3 Gradient Descent

We use Gradient descent as our baseline method for optimising our loss function. Gradient descent has been the standard approach to iteratively improving parameters in models for some time. We can define gradient descent as,

$$\theta_{k+1} = \theta_k - \alpha_k \nabla L(\theta_k)$$

This works due to the idea that the negated gradient direction gives the greatest reduction in the loss function, $L(\theta)$ per unit change of parameters, $\theta$.

We can acquire some motivation from a 1-order taylor series for $L(\theta_k)$,

$$L(\theta_k + h) \approx L(\theta_k) + \nabla L(\theta_k)' h$$
$$min_h\ L(\theta_k) + \nabla L(\theta_k)' h$$
$$min_h\ ||h|| ||\nabla L(\theta_k)|| cos(\phi),\ ||h|| = 1,\ cos(\pi) = -1$$
$$h = -\frac{\nabla L(\theta_k)}{||\nabla L(\theta_k)||}$$

Problems with gradient descent can be seen when considering a 2-dimensional quadratic. With a fixed step length, $\alpha_k$, for large step sizes, $\alpha_k$, it is possible to easily overshoot the minimiser and effectively miss the optimal solution. For small step sizes, $\alpha_k$, it is clear to see that as we reach the minimum, the steps taken will become continually small before reaching the minimiser.

We can show a more technical explanation of this failure by showing the gradient descent minimisation to a quadratic approximation to $L(\theta)$.

$$L(\theta_k + h) \approx L(\theta_k) + \nabla L(\theta_k)' h + \frac{1}{2} h' \nabla^2 L(\theta_k) h$$
$$\approx L(\theta_k) + \nabla L(\theta_k)' h + \frac{1}{2} h' (mI) h = L(\theta_k) + \nabla L(\theta_k)' h + \frac{m}{2} ||h||^2$$
$$argmin_h L(\theta_k) + \nabla L(\theta_k)' h + \frac{m}{2} ||h||^2 = -\frac{1}{m} \nabla L(\theta)$$

Approximating $\nabla^2 L(\theta)$ with $mI$, where $m$ is an arbitrary scalar, and $I$ is an identity matrix, this gives an approximation to the Hessian that gives all directions the same very high curvature which reveals some insight into why this method struggles on problems where curvature varies greatly.

Finally, we can take a look at the convergence rate of the method to further motivate our study of higher order methods, let $\theta^*$ be a local minimiser, and a fixed step size $t \leq \frac{1}{C}$, and $C$ was our Lipschitz constant defined above.

$$L(\theta_k) - L(\theta^*) \leq \frac{||\theta_0 - \theta^*||^2}{2tk}, \; t \leq \frac{1}{C}$$

where $C$ is the Lipschitz constant. This implies that gradient descent has a convergence rate of $\mathcal{O}(\frac{1}{k})$.

In summary, we know that advantages of gradient descent are that it is simple to implement and computing the gradient is a cheap operation at each iteration. However, it is often slow due to the fact that most problems are not strongly convex and its zig-zagging nature.

**Algorithm 3:** Gradient Descent with Line Search

**Result:** $\theta^*$

**while** *not converged* **do**
  $p_k = -\nabla L(\theta_k)$;
  $\alpha_k \leftarrow lineSearch(\cdot)$;
  $\theta_k = \theta_k + \alpha_k p_k$ ;
**end**
$\theta^* = \theta_k$;

## 3.4 Limited-Memory Broyden–Fletcher–Goldfarb–Shanno

We introduce a limited-memory variant of a Quasi-Newton method in order to potentially increase the speed of reducing the loss in terms of number of iterations.

In the original BFGS, each iteration update had the form, $x_{k+1} = x_k - \alpha_k H_k \nabla f_k$, where $\alpha_k$ was the step length and the inverse curvature matrix, $H_k$, was updated after every iteration, $H_{k+1} = V_k' H_k V_k + \rho_k s_k s_k'$. We define $\rho_k = \frac{1}{y_k' s_k}$ and $V_k = I - \rho_k y_k s_k'$, with $s_k = x_{k+1} - x_k$, $y_k = \nabla f_{k+1} - \nabla f_k$.

Clearly, the storage of the inverse Hessian and its manipulation is completely unreasonable for any practical deep neural network. In fact, any form of storage of a Hessian-like matrix is typically intractable for these problems and deters actual use of second-order optimisation methods.

The original BFGS algorithm consists of low rank (rank 2) updates to the constructed inverse approximate Hessian. By utilising LBFGS, we hope to also exploit some of this nature and attain a superlinear convergence rate in the optimisation. I expect that this method, while more computationally expensive than Gradient Descent, will perform better in terms of number of iterations.

The Limited-Memory Broyden–Fletcher–Goldfarb–Shanno(LBFGS) is a method that approximates the inverse Hessian by only considering a 'history', $m$, of previous differences in iterates, $s_k$, and gradients,$y_k$ in order to compute just an approximate Hessian-vector product through summations and inner products alone. The time complexity of this algorithm is $\mathcal{O}(mn)$, where $n$ is

the number of variables, and $m$ is the window size.

**Algorithm 4:** L-BFGS two-loop recursion

**Result:** $H_k \nabla L(\theta_k) = r$

$q \leftarrow \nabla L(\theta_k)$;
**for** $i = k-1,\ ...,\ k-m$ **do**
  $\alpha_i \leftarrow \rho_i s_i' q$ ;
  $q \leftarrow q - \alpha_i y_i$;
**end**
$r \leftarrow H_k^0 q$;
**for** $i = k-m,\ ...,\ k-1$ **do**
  $\beta \leftarrow \rho_i y_i' r$;
  $r \leftarrow r + s_i(\alpha_i - \beta)$;
**end**

**Algorithm 5:** L-BFGS

**Result:** $\theta^*$

**while** *not converged* **do**
  $H_k^0 \leftarrow \frac{s_{k-1}' y_{k-1}}{y_{k-1}' y_{k-1}} I$;
  $p_k \leftarrow -H_k \nabla L(\theta_k)$;
  $\alpha_k \leftarrow lineSearch(\cdot)$;
  $\theta_k = \theta_k + \alpha_k p_k$ ;
  **if** $k > m$ **then**
    discard $s_{k-m},\ y_{k-m}$;
  **end**
  SAVE $s_k \leftarrow L(\theta_{k+1}) - L(\theta_k)$;
  SAVE $y_k \leftarrow \nabla L(\theta_{k+1}) - \nabla L(\theta_k)$;
**end**
$\theta^* = \theta_k$;

### 3.5 Newton Conjugate Gradient

The line search Newton conjugate gradient method is also known as the truncated Newton method. The search direction is computed by applying conjugate gradients to the Newton equations, $\nabla^2 L_k p_k = -\nabla L_k$. However, the Hessian may have negative eigenvalues, and if this is the case we terminate the CG iteration. This modification produces a guaranteed descent direction. If we replace, the Hessian with $B_k$, when it is positive definite, we should expect a newton-like step, and a gradient-step when there is negative curvature detected. With these conditions, this method has superlinear convergence and should perform better than Gradient Descent and similarly to LBFGS. The time complexity associated with this method is $\mathcal{O}(s\sqrt{k})$ where $s$ is the number of

non-zero entries in $B_k$ and $k$ is the condition number.

**Algorithm 6:** Line Search Newton-CG

**Result:** $\theta^*$

**while** *not converged* **do**
  $\epsilon = min(0.5, \sqrt{||\nabla L_k||})||\nabla L_k||$;
  $z_0 = 0,\ r_0 = \nabla L_k,\ d_0 = -r_0$;
  **for** *until MaxIterations* **do**
    **if** $d_j' B_k d_j \leq 0$ **then**
      **if** $j = 0$ **then**
        $p_k = -\nabla L_k$;
      **else**
        $p_k = z_j$;
      **end**
      $\alpha_j = \frac{r_j' r_j}{d_j' B_k d_j}$;
      $z_{j+1} = \alpha_j d_j$;
      $r_{j+1} = r_j + \alpha_j B_k d_j$;
      **if** $||r_{j+1}|| < \epsilon$ **then**
        $p_k = z_{j+1}$;
      **end**
      $\beta_{j+1} = \frac{r_{j+1}' r_{j+1}}{r_j' r_j}$;
      $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$;
    **end**
  **end**
  $\alpha_k \leftarrow lineSearch(\cdot)$;
  $\theta_k = \theta_k + \alpha_k p_k$ ;
**end**
$\theta^* = \theta_k$;

For the Hessian-vector product computation, we make use of a finite differences approach where such that we can attain a Hessian vector product.

$$\nabla^2 B_k d \approx \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h},\ h \approx 10e - 8$$

### 3.6 Newton-Steihaug Conjugate Gradient

In this method, we modify the above method so that it uses a trust region approach. This method still retains global superlinear convergence, but should be able to fix problem in when the Hessian is nearly singular, which can cause many evaluations during line search when looking for a step

size in the original truncated Newton method.

**Algorithm 7:** CG-Steihaug

**Result:** $\theta^*$
**while** *not converged* **do**
  $\epsilon > 0$;
  $z_0 = 0,\ r_0 = \nabla L_k,\ d_0 = -r_0$;
  **if** $||r_0|| < \epsilon$ **then**
    $p_k = z_0$;
  **end**
  **for** *until MaxIterations* **do**
    **if** $d_j' B_k d_j \leq 0$ **then**
      Find $\tau$ s.t. $p_k = z_j + \tau d_j$ minimises the model and $||p_k|| = \Delta$;
    **end**
    $\alpha_j = \frac{r_j' r_j}{d_j' B_k d_j}$;
    $z_{j+1} = \alpha_j d_j$;
    **if** $||z_{j+1}|| \geq \Delta$ **then**
      Find $\tau \geq 0$ s.t. $p_k = z_j + \tau d_j$ and $||p_k|| = \Delta$;
    **end**
    $r_{j+1} = r_j + \alpha_j B_k d_j$;
    **if** $||r_{j+1}|| < \epsilon$ **then**
      $p_k = z_{j+1}$;
    **end**
    $\beta_{j+1} = \frac{r_{j+1}' r_{j+1}}{r_j' r_j}$;
    $d_{j+1} = -r_{j+1} + \beta_{j+1} d_j$;
  **end**
  $\theta_k = \theta_k + p_k$ ;
**end**
$\theta^* = \theta_k$;

# 4 Case Studies

Hyperparameters chosen in the following case studies were chosen by inspection and chosen to best highlight key differences in the various optimiser approaches as well as recommended settings for line search and trust region methods.

It should be noted that in the backpropagation process, we have added a functionality that terminates the updating of weight matrices if their percent change is within a specified tolerance. If a weight matrix update is within this tolerance, we consider these weights to be sufficiently learned for plot and comparison purposes. However, this does affect the interplay between weight matrices. We have specified this tolerance in table below.

It should also be noted that the current implementation of the CG-Steihaug method does not work entirely as intended and therefore, we disregard it in the analysis of the plots provided.

These plots are logarithmically scaled, so the difference in loss functions and step sizes could be seen clearly.

**More plots are left in the ipython notebook for if every plot were included the document would be over 60 pages in length.**

## 4.1 Varying Depth of Networks

In this study, we examine the performance of these optimisers on each of the weight matrices during the backpropagation process of training deep neural networks as we vary the number of layers in a network. For these experiments, the settings of the parameters are,

| - | **Gradient Descent** | **L-BFGS** | **Newton CG** | **CG-Steihaug** |
|---|---|---|---|---|
| Layers | 1,2,3,4,5 | 1,2,3,4,5 | 1,2,3,4,5 | 1,2,3,4,5 |
| Units/Hidden Layer | 20 | 20 | 20 | 20 |
| Hidden Activation Function | Sigmoid | Sigmoid | Sigmoid | Sigmoid |
| Output Activation Function | Softmax | Softmax | Softmax | Softmax |
| Batch Size | 100 | 100 | 100 | 100 |
| Max Inner Iteration | - | - | 100000 | 100000 |
| Epochs | 1 | 1 | 1 | 1 |
| Step Method | Backtrack LS | Backtrack LS | Backtrack LS | Trust Region |
| Trust, $\eta$ | - | - | - | 5.e-4 |
| $h$ | 10.e-8 | 10.e-8 | 10.e-8 | 10.e-8 |
| $m$ | - | 5 | - | - |
| Trust, $\Delta_0$ | - | - | - | 1 |
| Line Search, $\alpha_0$ | 1 | 1 | 1 | - |
| Line Search, $c_1$ | 1.e-4 | 1.e-4 | 1.e-4 | - |
| Line Search, $\rho$ | 0.8 | 0.8 | 0.8 | - |
| tolerance | 1.e-14 | 1.e-14 | 1.e-14 | 1.e-14 |

Analysing at our results, we will first examine the training accuracy.

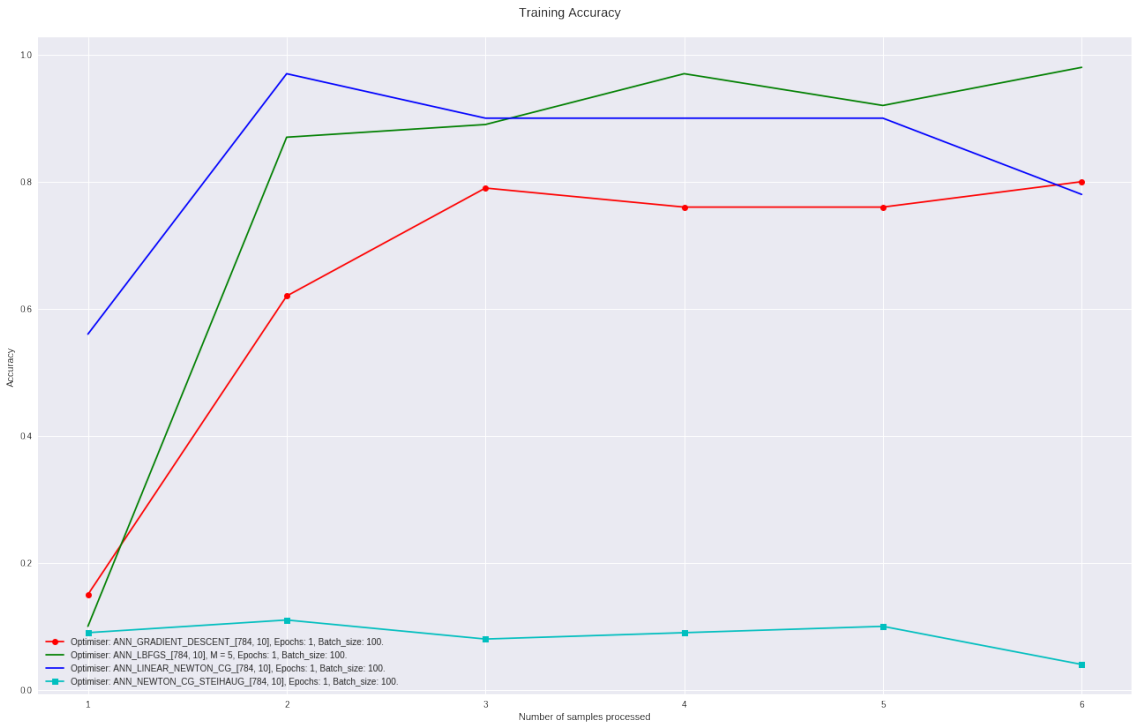


Figure 3: 1 Layer Neural Network Training Accuracy.

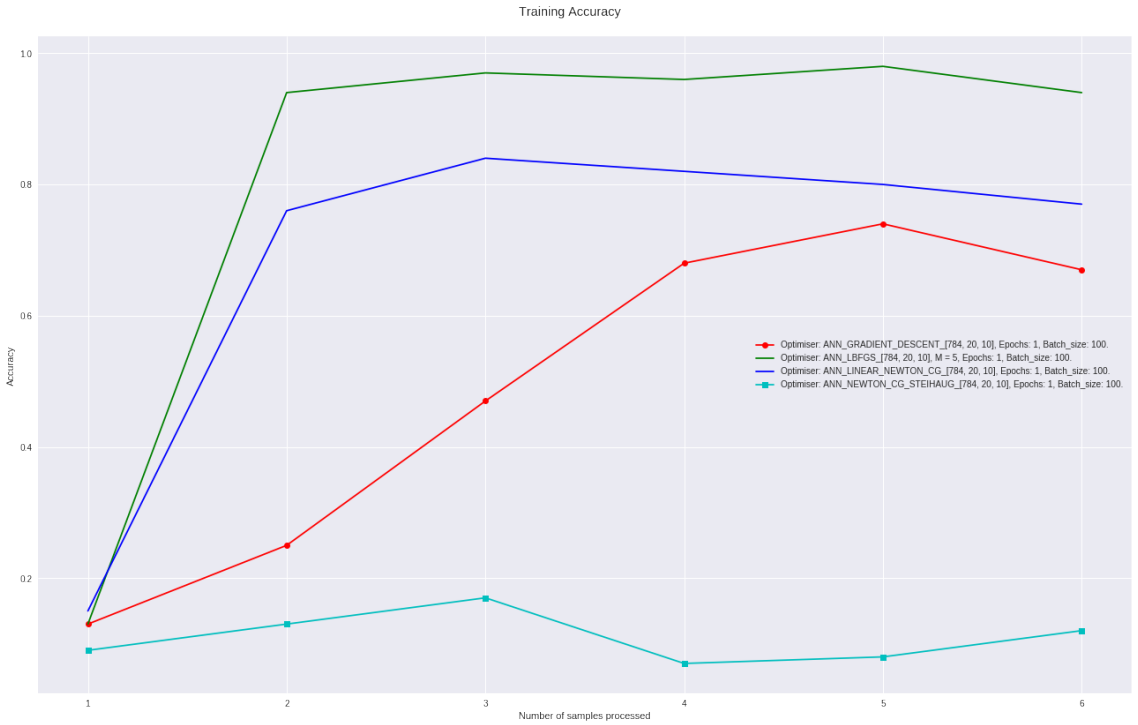


Figure 4: 2 Layer Neural Network Training Accuracy.

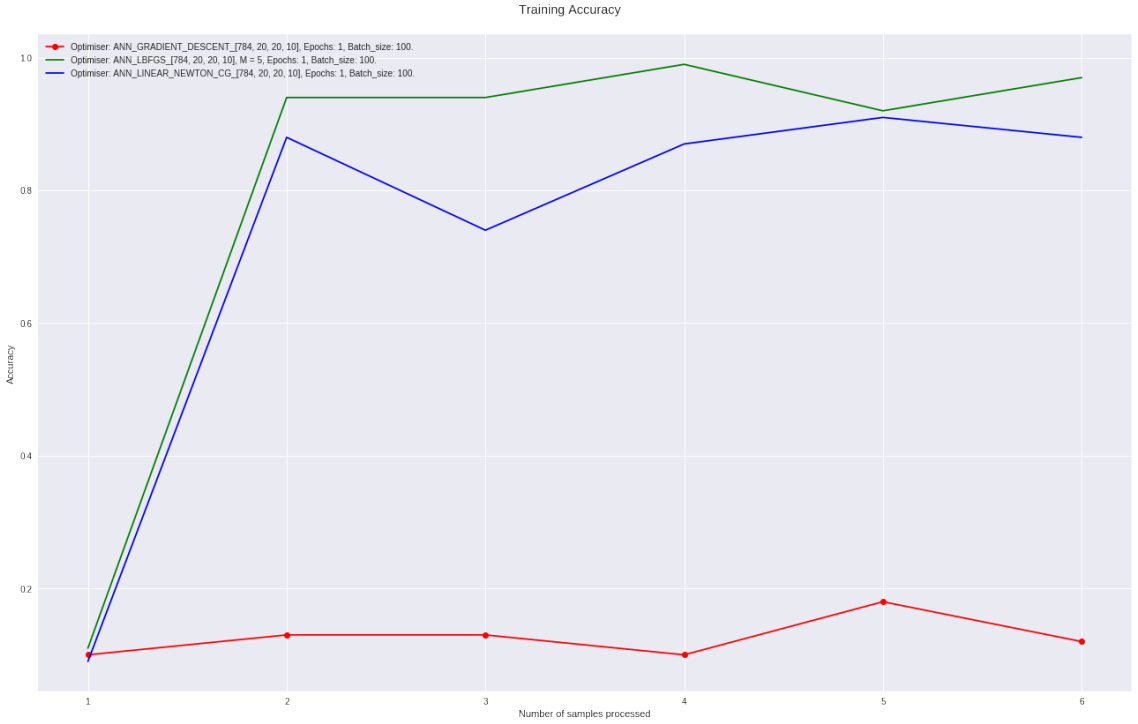


Figure 5: 3 Layer Neural Network Training Accuracy.

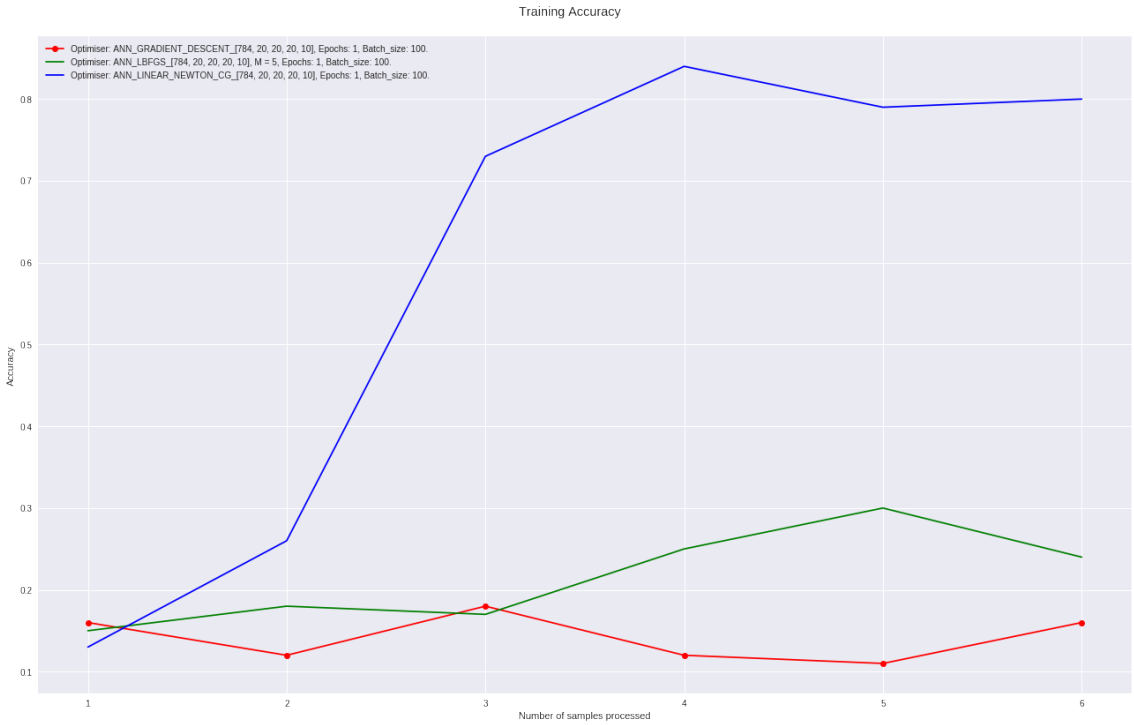


Figure 6: 4 Layer Neural Network Training Accuracy.

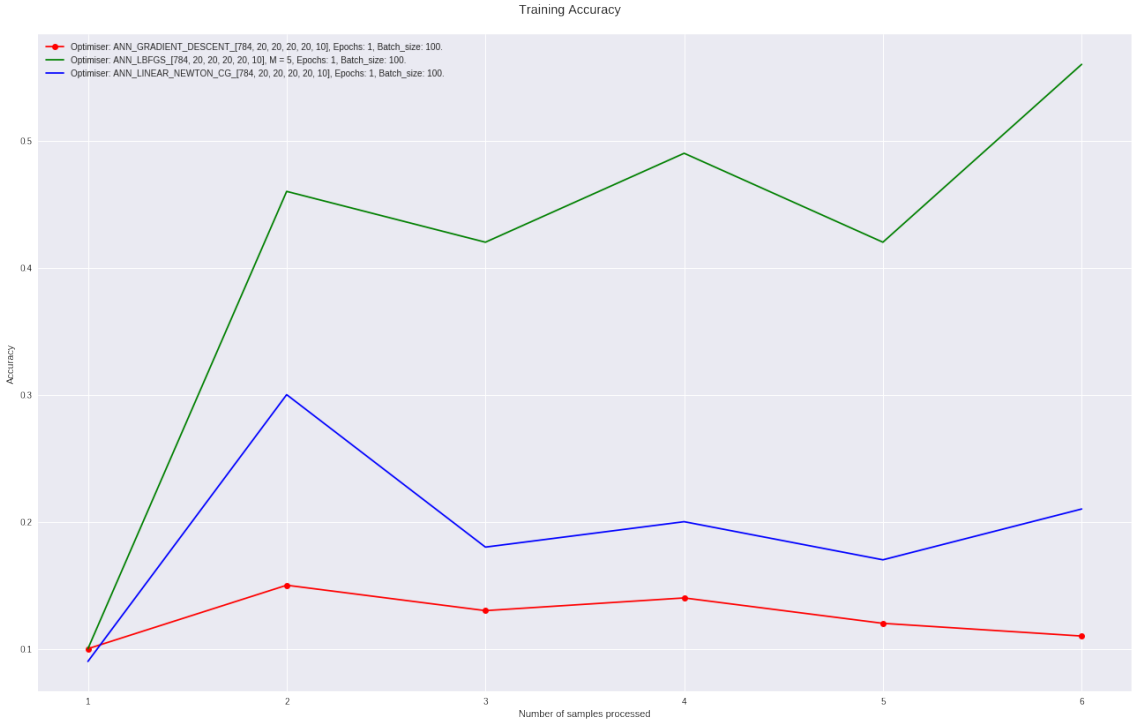


Figure 7: 5 Layer Neural Network Training Accuracy.

From the plots, it is clear to see that their is a trend in that as the number of layers increases with a fixed batch size, the stochastic gradient descent method tends to perform worse than the respective quasi newton, and conjugate gradient based methods. These methods almost always achieve higher than 80 percent training accuracy as compared to the subpar gradient descent based method over a single epoch.

Moving onto the plots of the cross entropy loss curves, which was our objective to minimise.

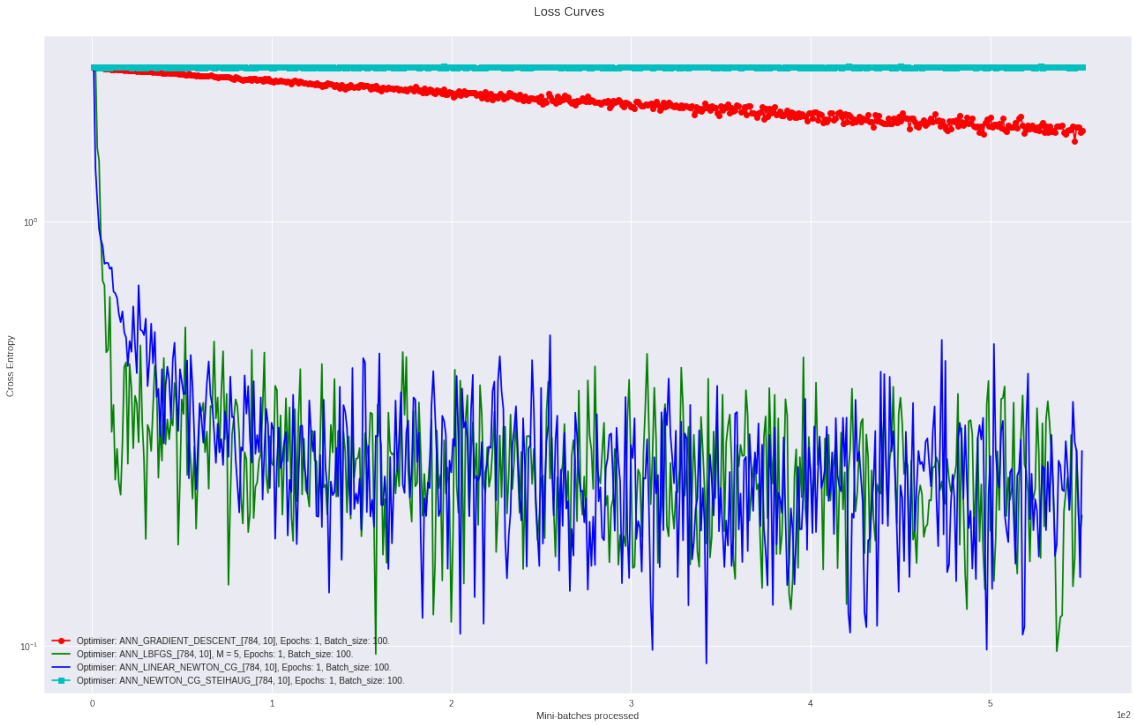


Figure 8: 1 Layer Neural Network Loss.

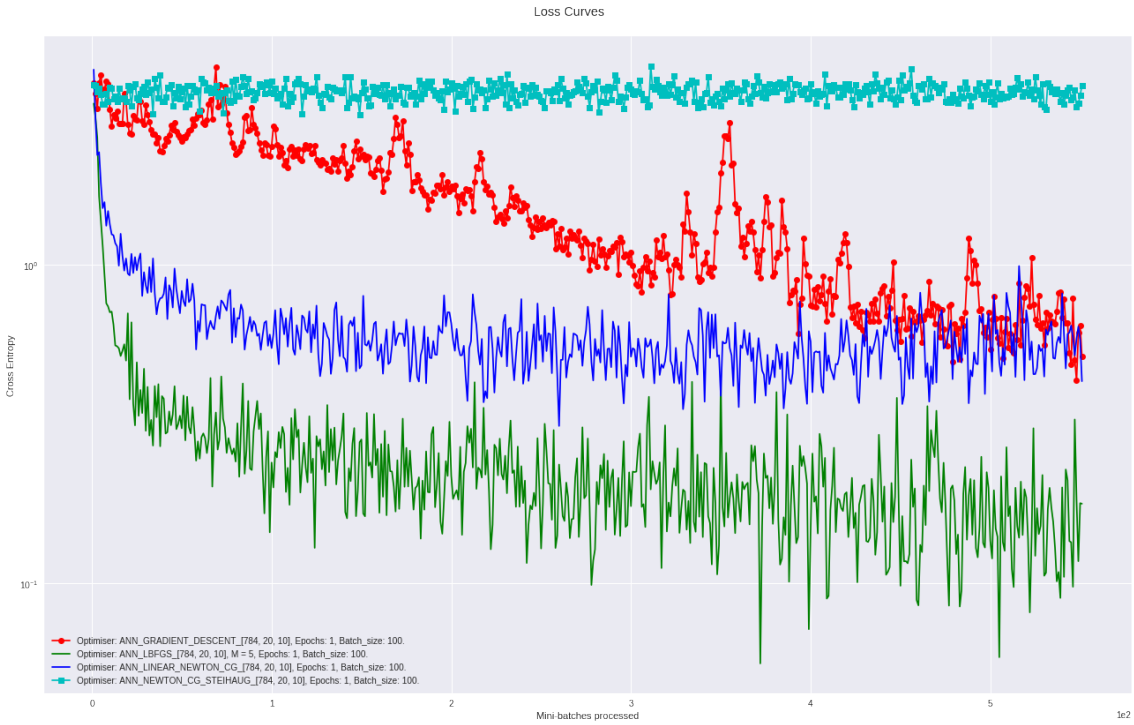


Figure 9: 2 Layer Neural Network Loss.

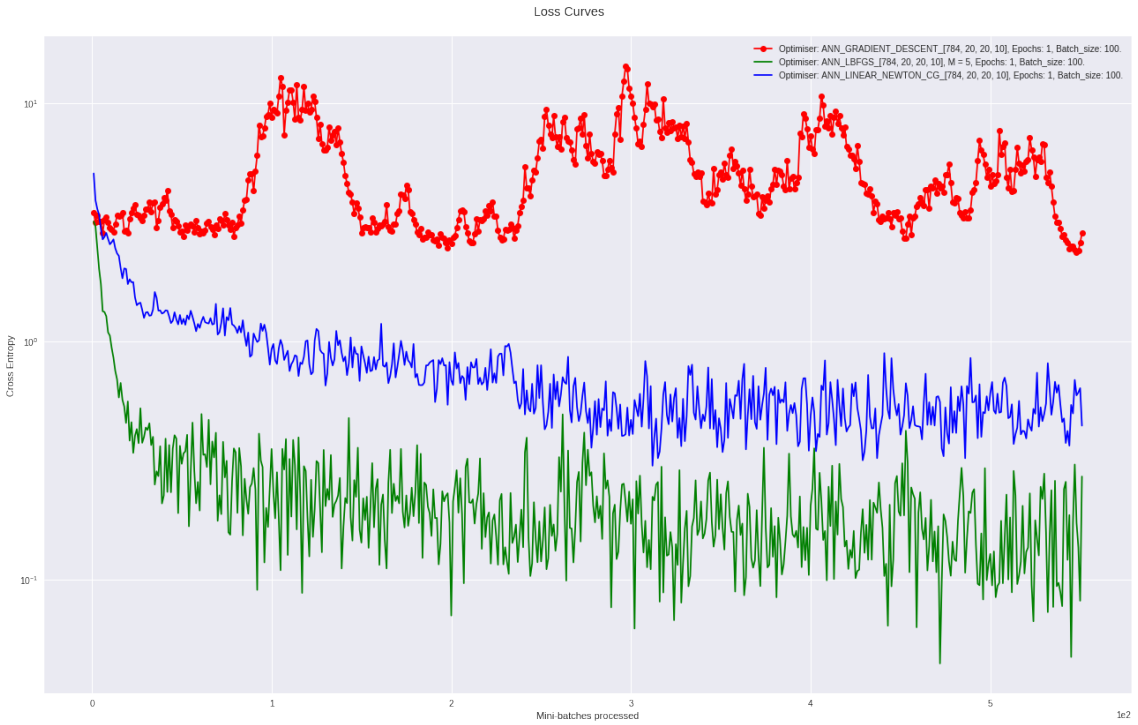


Figure 10: 3 Layer Neural Network Loss.

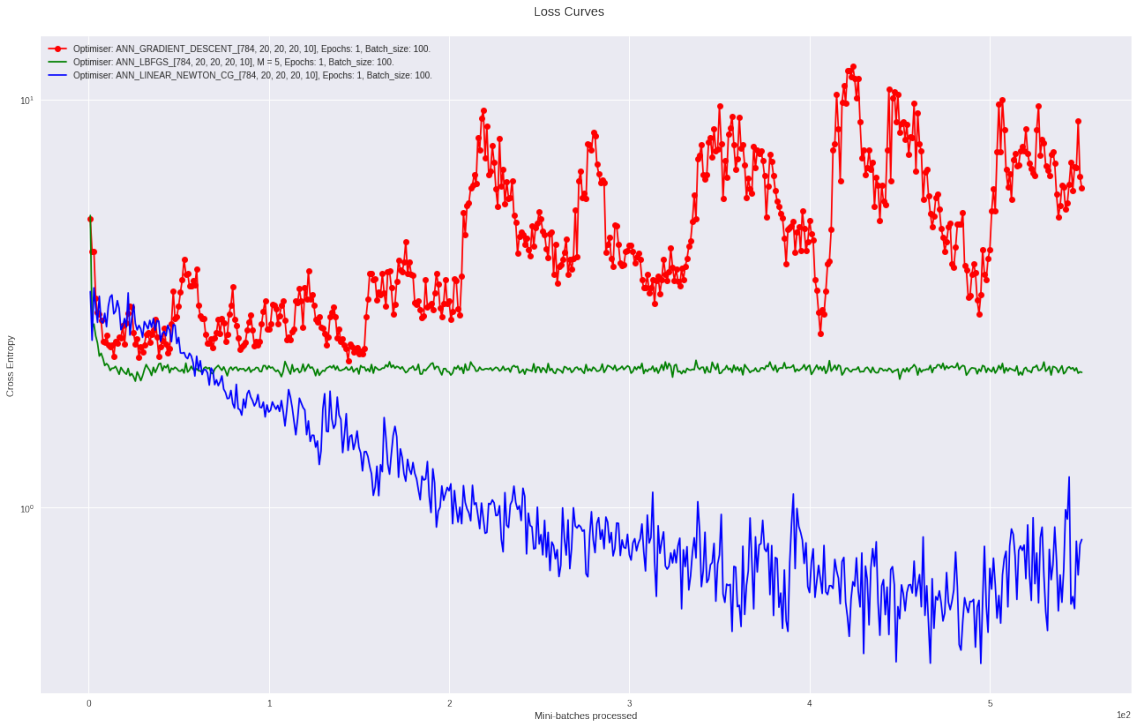


Figure 11: 4 Layer Neural Network Loss.

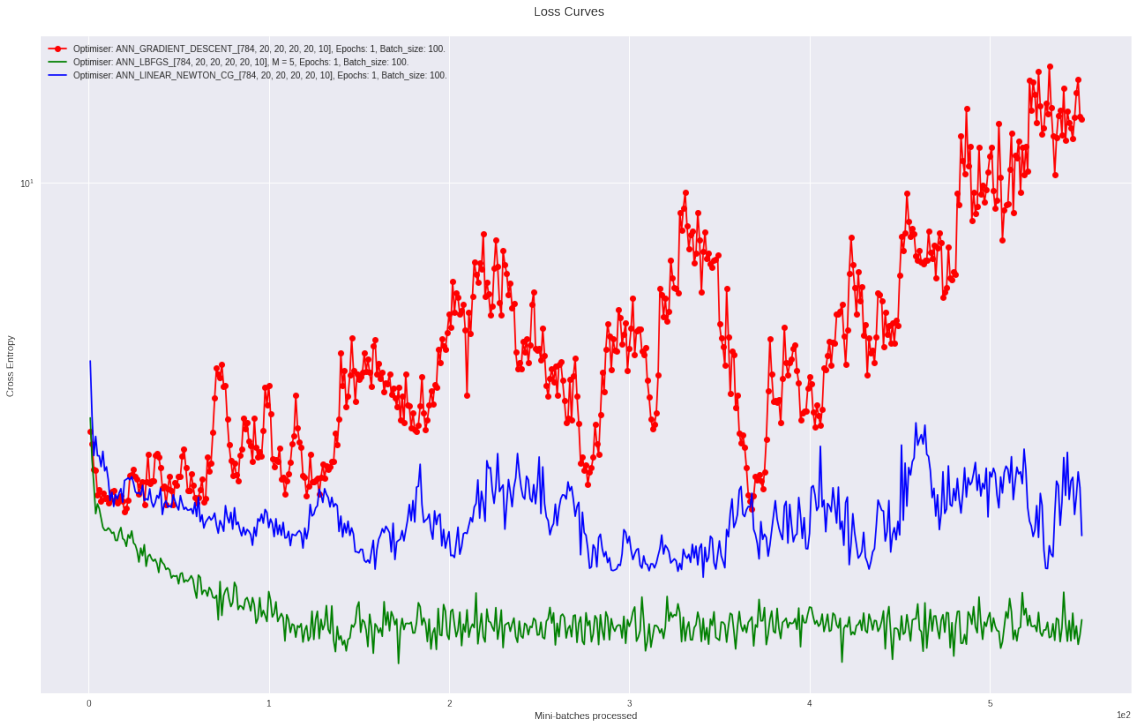


Figure 12: 5 Layer Neural Network Loss.

From interpreting these plots, it is very clear that as we increase the number of layers the L-BFGS approach with a window of $M = 5$ is the clearly superior method in terms of reduction of per mini-match iteration. It consistently achieves a more dramatic loss reduction curve. The Newton CG method performs well with a steady reduction of the loss function. However, interestingly, as we increase the number of layers, the gradient descent method begins to perform worse and even diverge. This is presumably due to the other fixed variables, the number of units per layers, batch size, and number of epochs. We can say that our approximations of the Hessian are typically quite good in comparison to just taking the gradient update.

Further investigating, we can look at the step sizes, $\alpha_k$, and full steps for the trust region method, $p_k$,

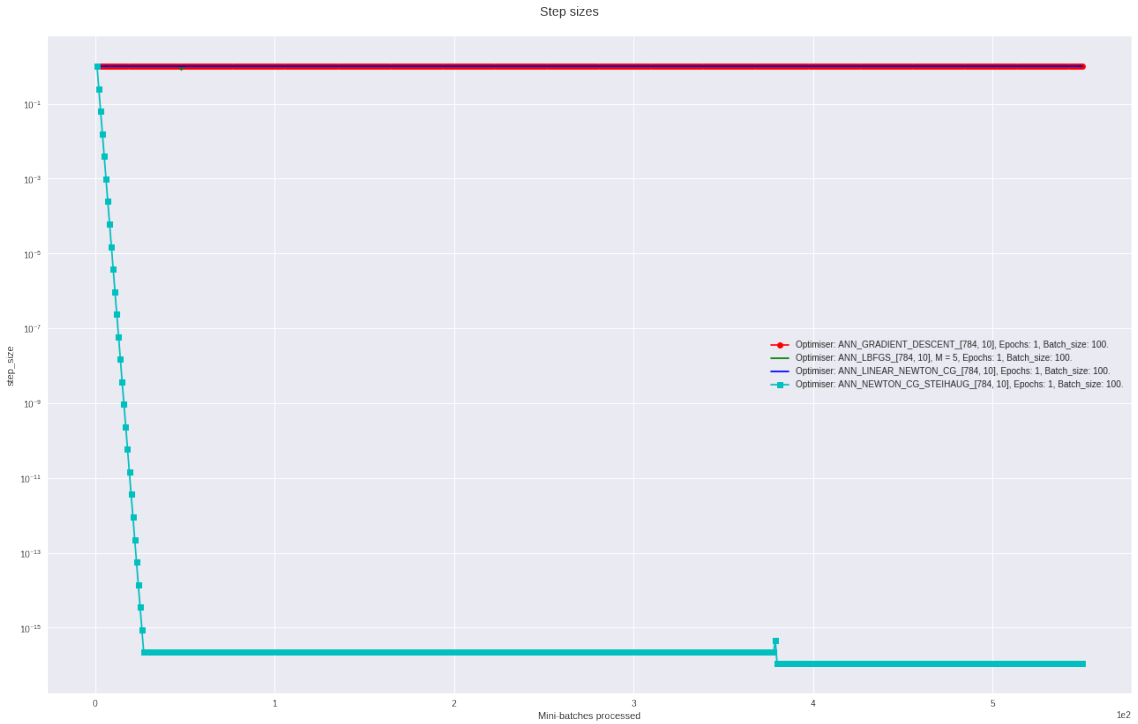


Figure 13: 1 Layer Neural Network Step sizes.

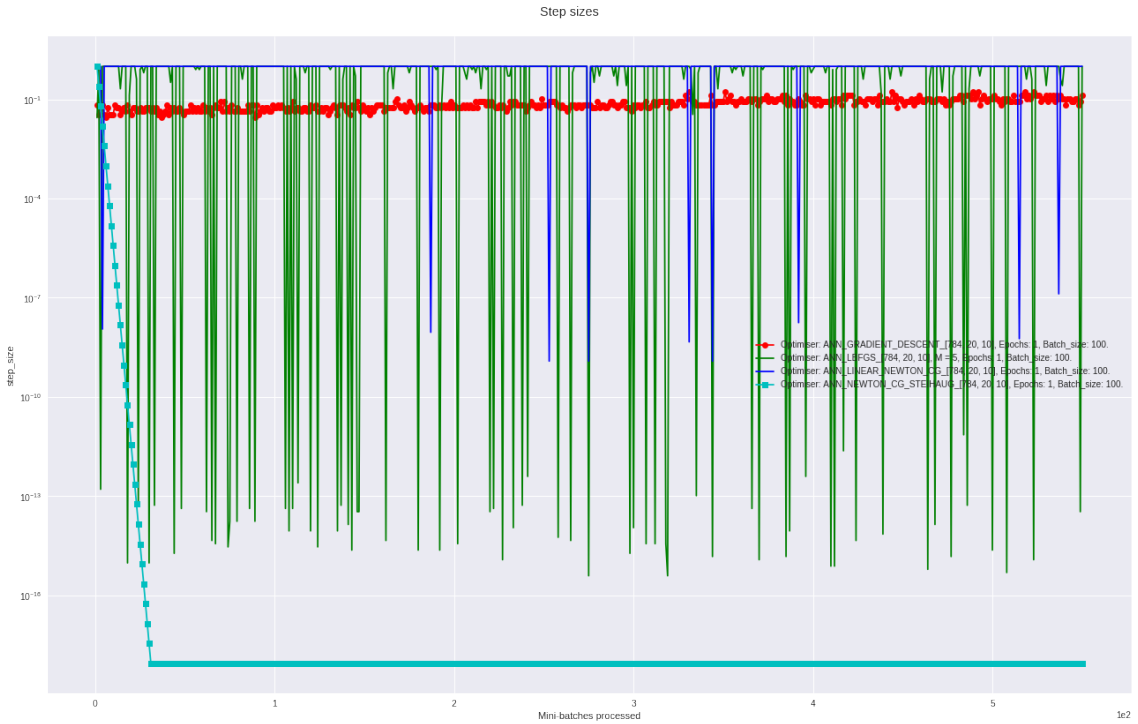


Figure 14: 2 Layer Neural Network Step sizes.

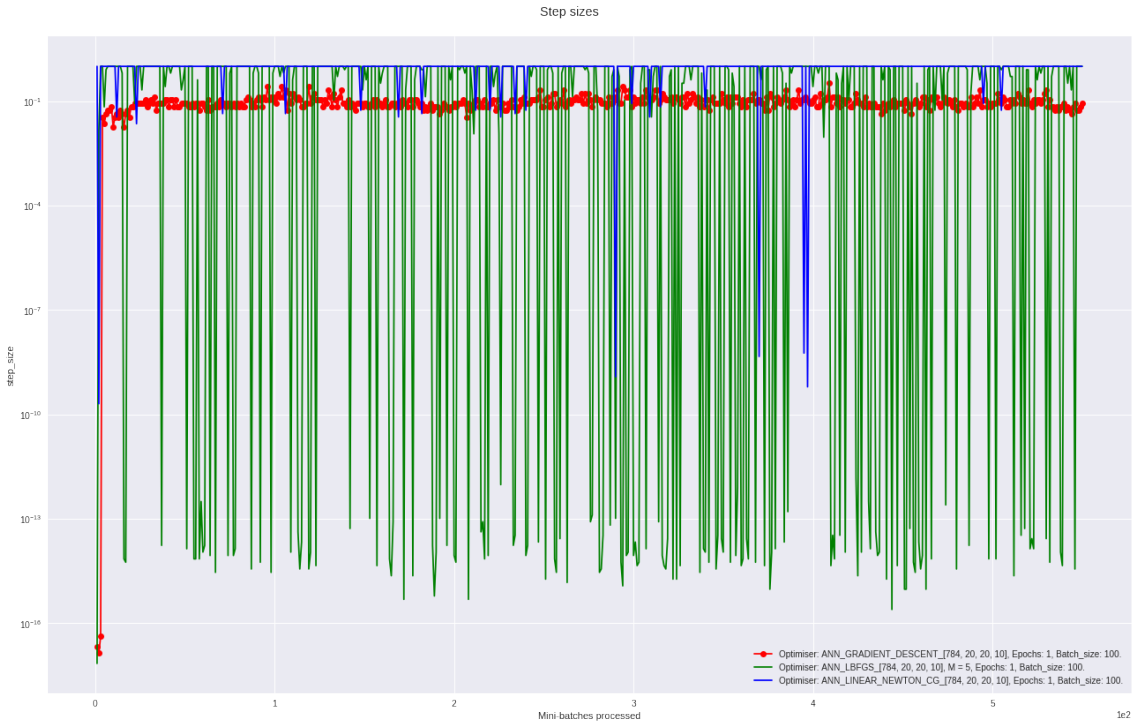


Figure 15: 3 Layer Neural Network Step sizes.

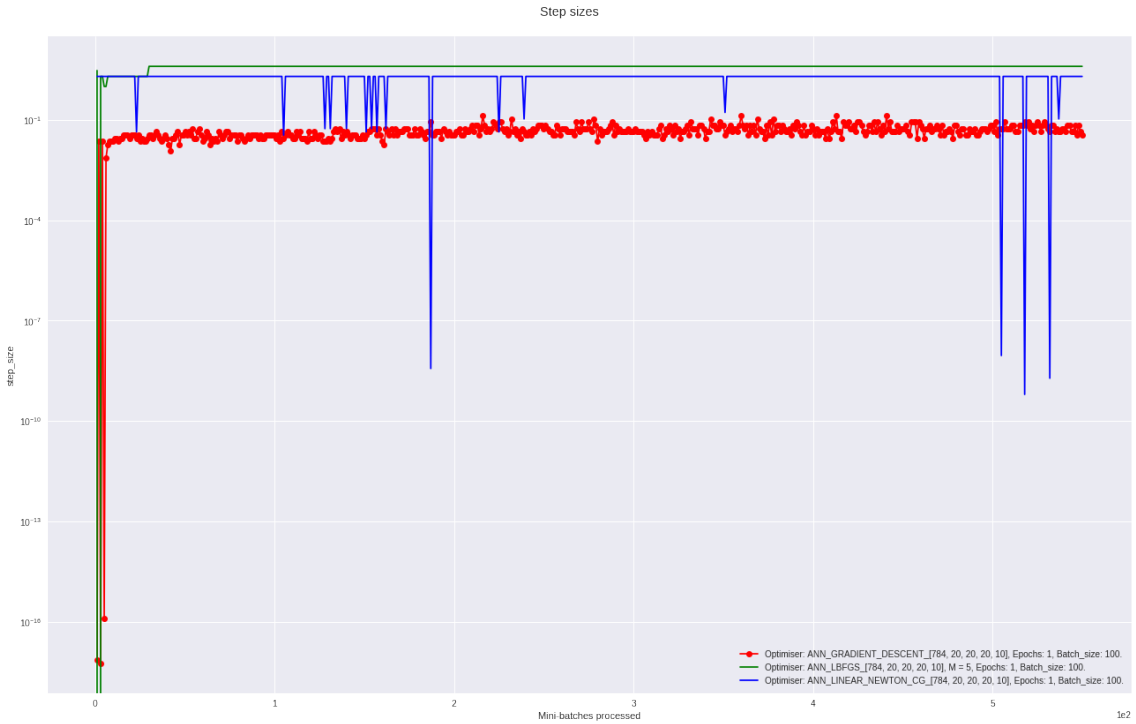


Figure 16: 4 Layer Neural Network Step sizes.

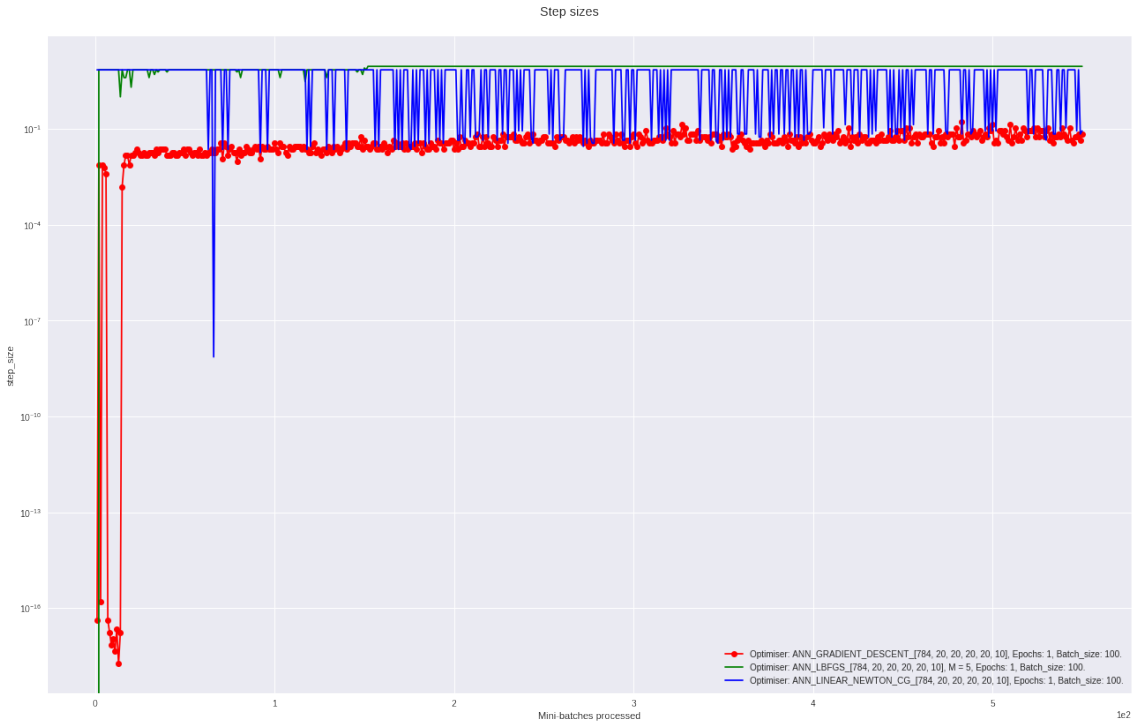


Figure 17: 5 Layer Neural Network Step sizes.

For the 2 Layer and 3 Layer networks, the volatility of the step sizes for the LBFGS method is much larger than the truncated newton and gradient descent methods. However, this changes and becomes much more stable for the 4 and 5 Layer networks. In general, the step size for the gradient descent is less than that of the Newton conjugate gradient and LBFGS methods. It could be argued that in this sense, our search directions generated for the mentioned methods are much better in terms of reduction than the gradient based method.

Finally, we will look at the number of iterations needed for convergence to within tolerance for each weight matrix according to each optimiser for 2 architecture settings,

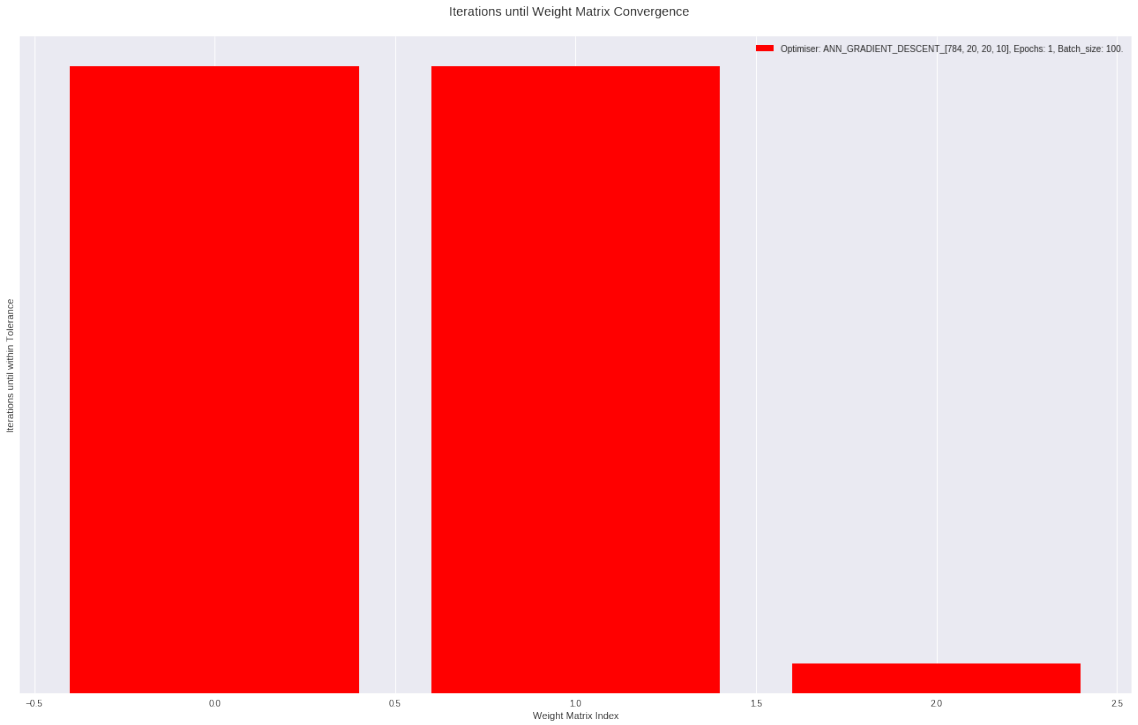


Figure 18: 3 Layer Neural Network Iteration for Weight Converge.

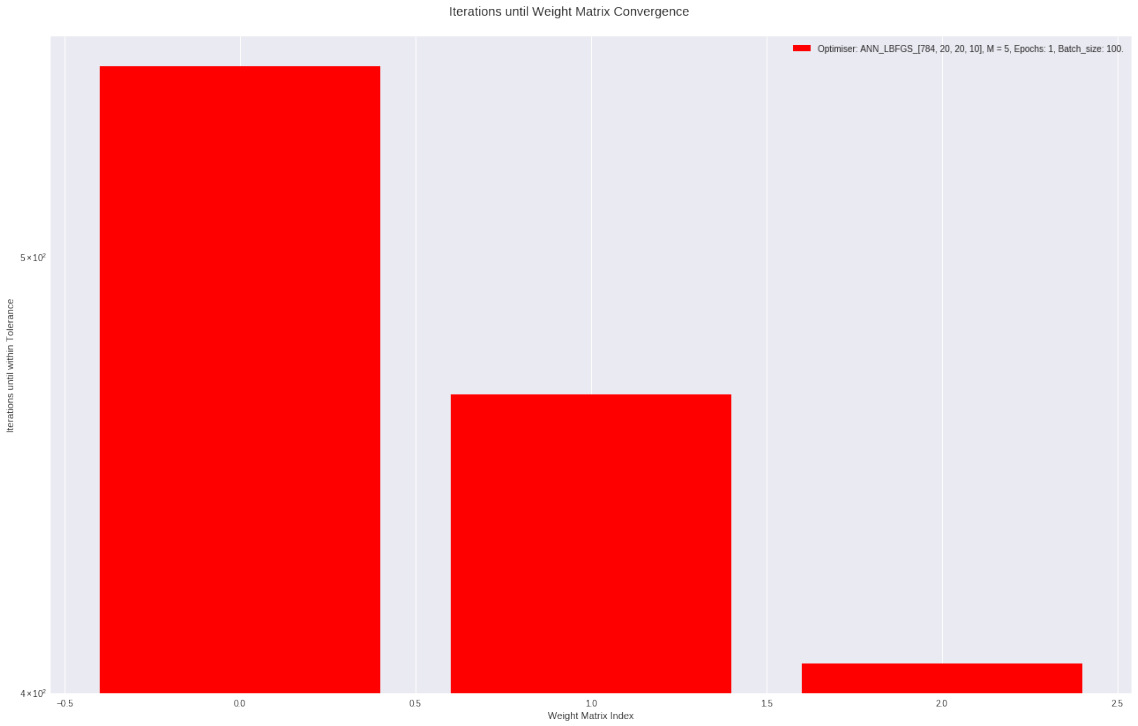


Figure 19: 3 Layer Neural Network Iteration for Weight Converge.

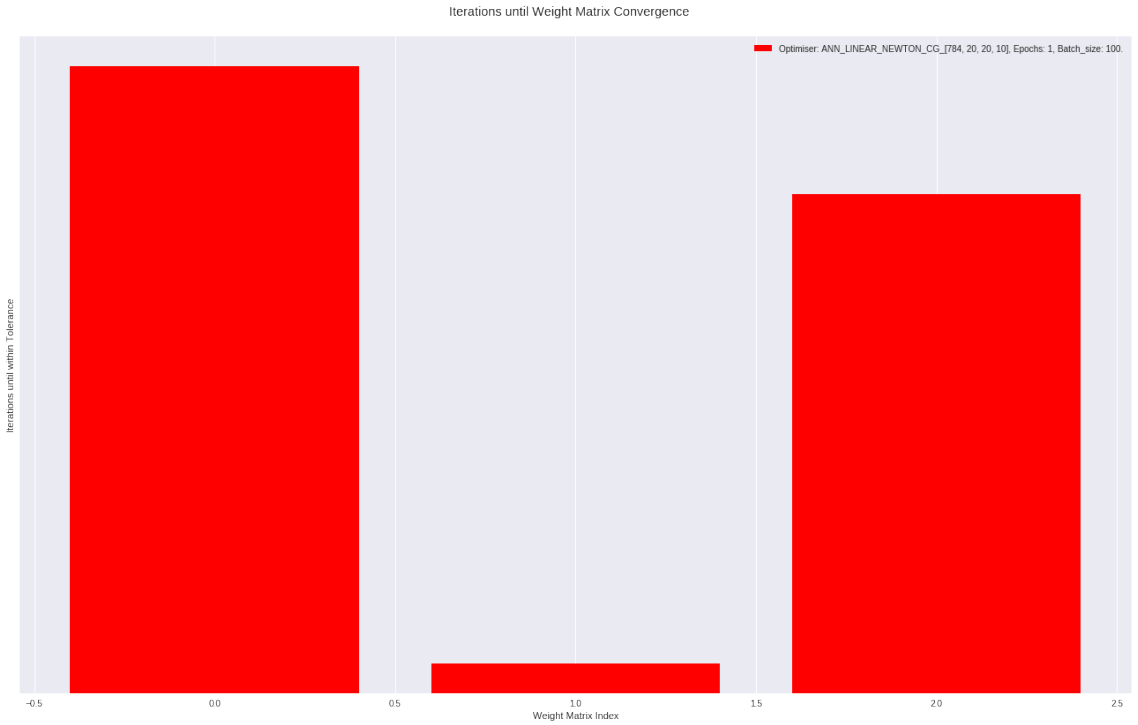


Figure 20: 3 Layer Neural Network Iteration for Weight Converge.

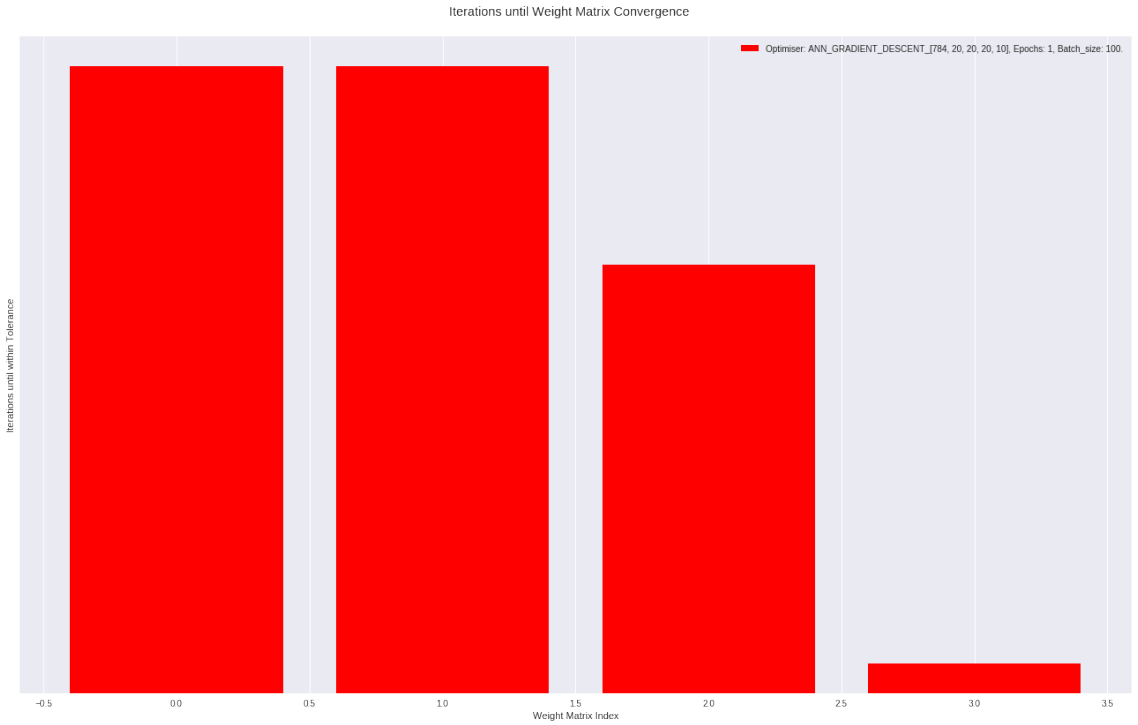


Figure 21: 4 Layer Neural Network Iteration for Weight Converge.

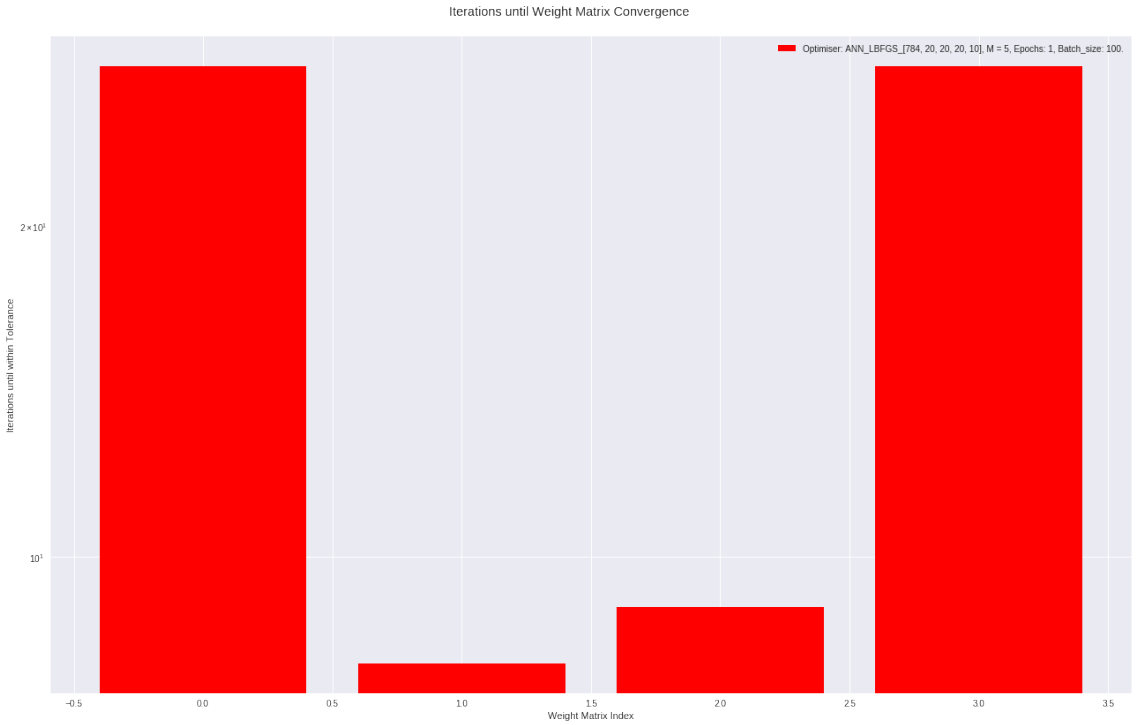


Figure 22: 4 Layer Neural Network Iteration for Weight Converge.

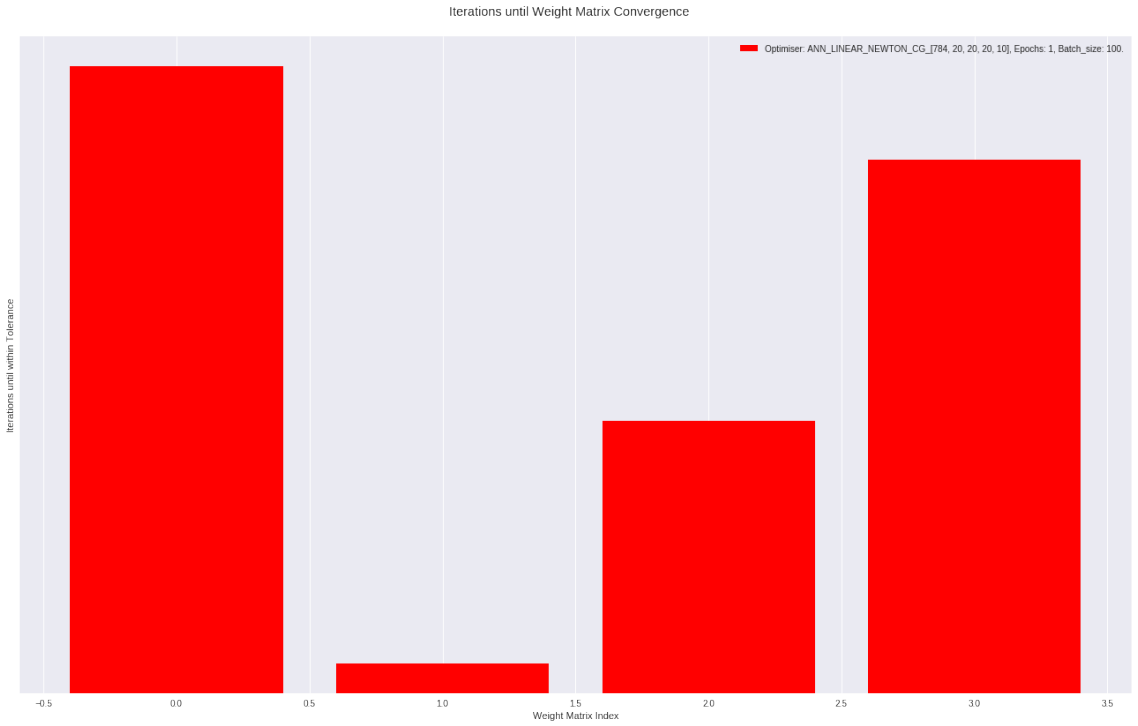


Figure 23: 4 Layer Neural Network for Weight Converge.

Each bar represents the number of iterations a weight matrix in the neural network needed to converge to within a tolerance. For example, in Figure 23, the first bar represents the matrix after the first layer, and the last bar represents the final weight matrix before output.

Interestingly, the final weight matrices for the gradient descent optimisers converged to their final values in fewer iterations than the weight matrices that are closer to the initial layers of the network. However, when we look at the LBFGS and Newton Conjugate Gradient method, the

weight matrices towards the middle of the networks are those that converge first. This is not shown here, but in the ipython notebook, this is even apparent for networks with even greater depth.

Perhaps this has to deal with the notion of smaller and smaller gradient as we backpropagate errors through the network. With larger gradient at the output of the network, we can make more informed steps that best represent the direction that weight matrix should travel in order to minimise the objective function.

As an addendum, I have included the steps needed for the inner iterations of the Newton conjugate gradient algorithm on the 4 Layer Network. There seems to be little trend in the iterations needed for a search direction across the number of total iterations, but the number of iterations needed for solving the newton equation does plateau at different levels for each of these matrices between 0 and 100.

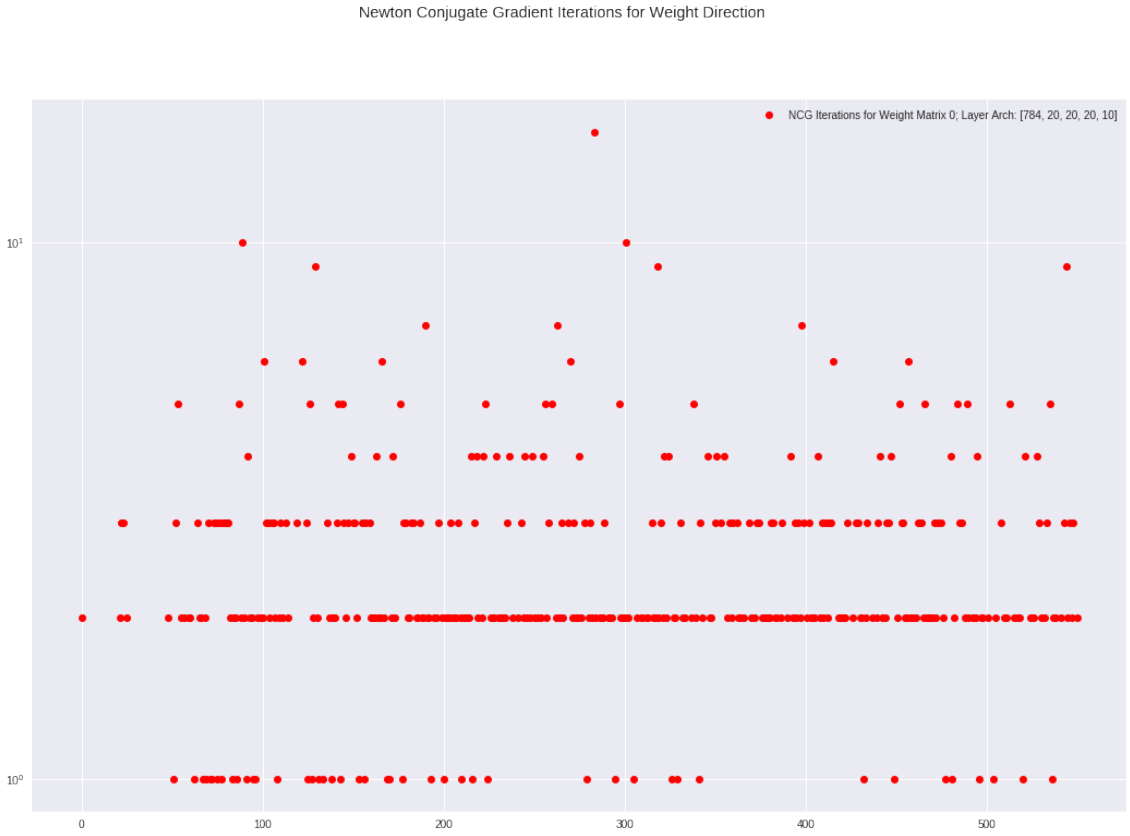


Figure 24: 4 Layer Neural Network NCG Iteration for Weight Matrix 0.

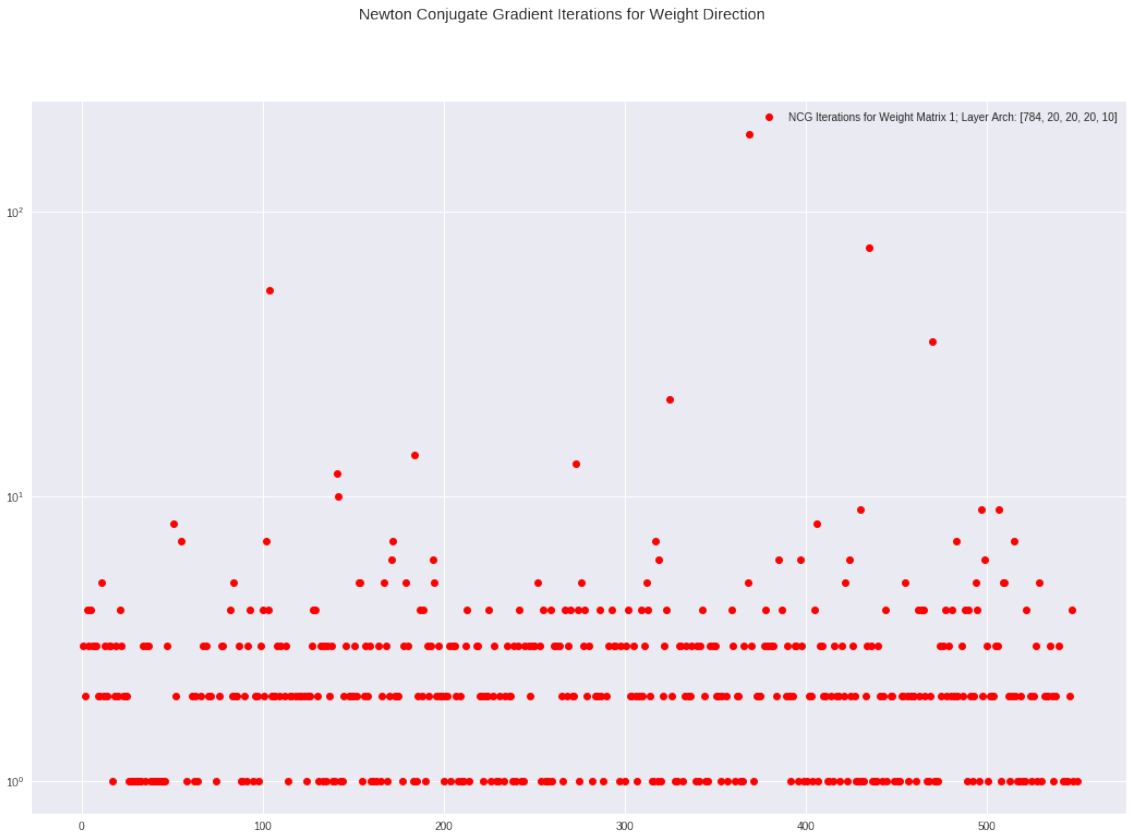


Figure 25: 4 Layer Neural Network NCG Iteration for Weight Matrix 1.

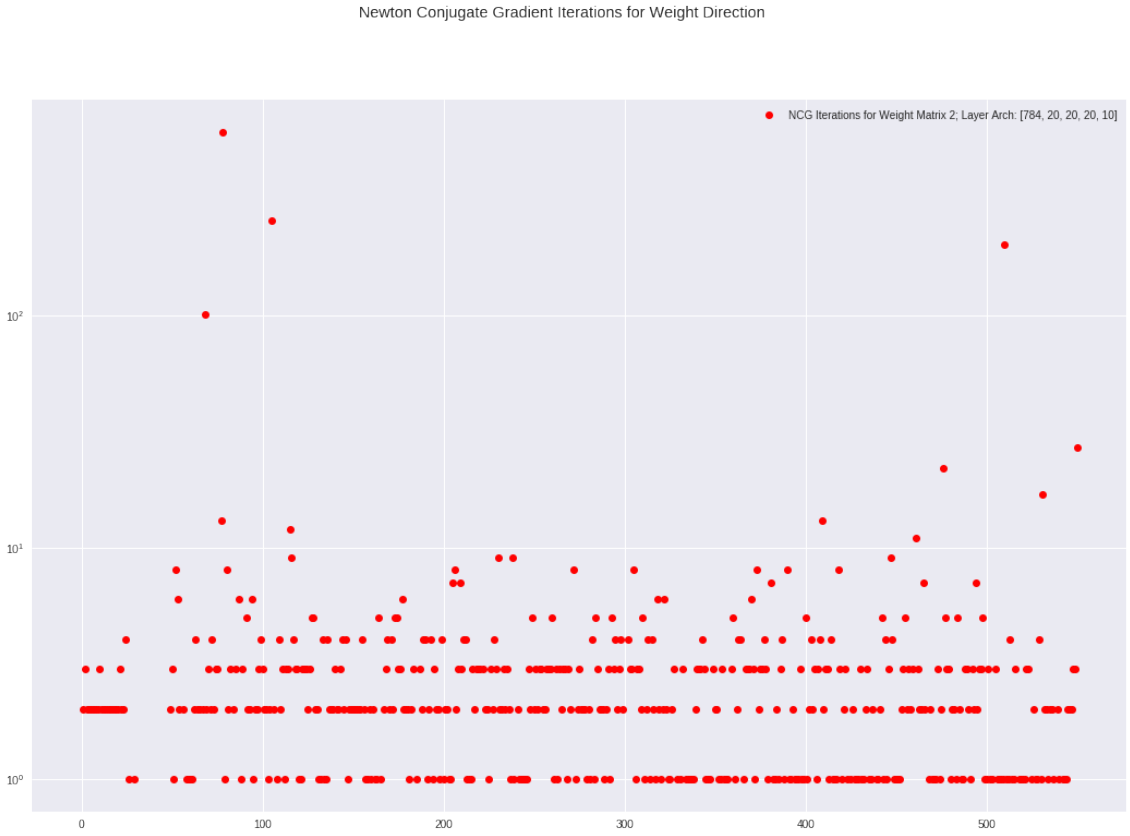


Figure 26: 4 Layer Neural Network NCG Iteration for Weight Matrix 2.

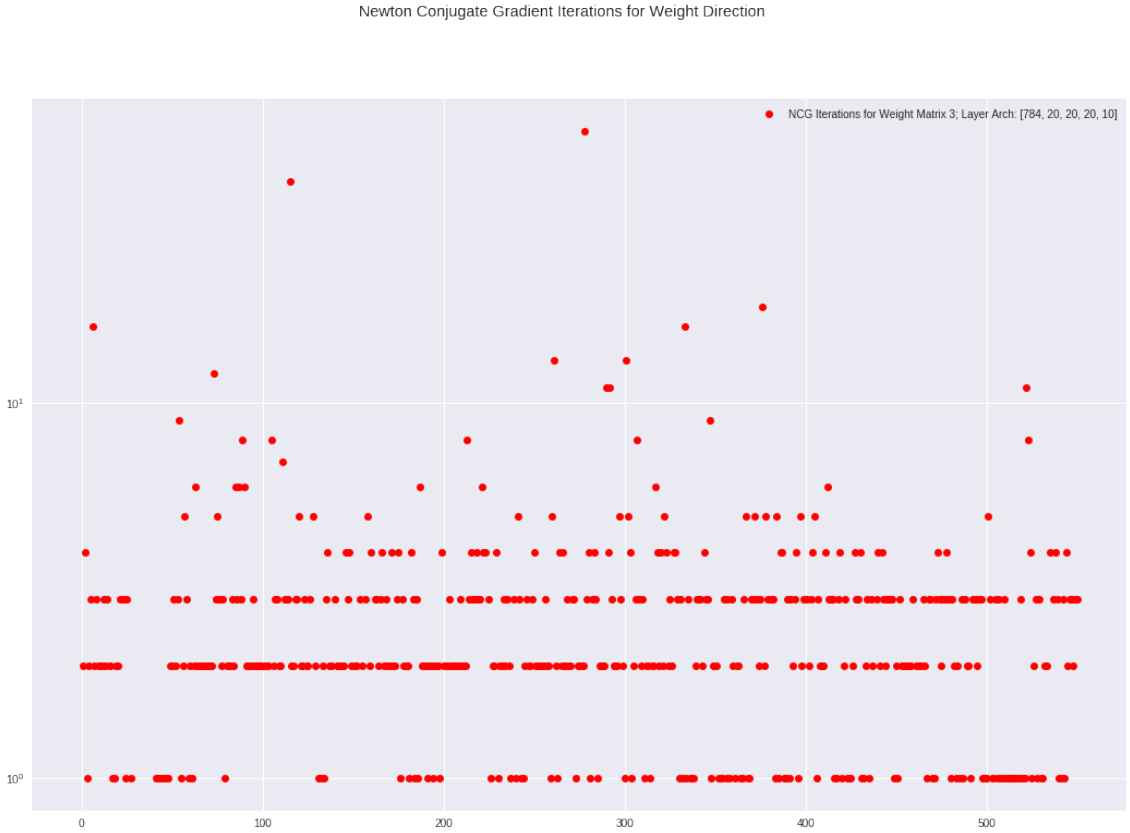


Figure 27: 4 Layer Neural Network NCG Iteration for Weight Matrix 3.

## 4.2 Varying Batch size of Networks

In this study, we examine the performance of these optimisers on each of the weight matrices during the backpropagation process of training deep neural networks as we vary the training mini-batch size. For these experiments, the settings of the parameters are,

| - | **Gradient Descent** | **L-BFGS** | **Newton CG** | **CG-Steihaug** |
|---|---|---|---|---|
| Layers | 2 | 2 | 2 | 2 |
| Units/Hidden Layer | 20 | 20 | 20 | 20 |
| Hidden Activation Function | Sigmoid | Sigmoid | Sigmoid | Sigmoid |
| Output Activation Function | Softmax | Softmax | Softmax | Softmax |
| Batch Size | 1,100,1000,10000 | 1,100,1000,10000 | 1,100,1000,10000 | 1,100,1000,10000 |
| Max Inner Iteration | - | - | 100000 | 100000 |
| Epochs | 1 | 1 | 1 | 1 |
| Step Method | Backtrack LS | Backtrack LS | Backtrack LS | Trust Region |
| Trust, $\eta$ | - | - | - | 5.e-4 |
| $h$ | 10.e-8 | 10.e-8 | 10.e-8 | 10.e-8 |
| $m$ | - | 5 | - | - |
| Trust, $\Delta_0$ | - | - | - | 1 |
| Line Search, $\alpha_0$ | 1 | 1 | 1 | - |
| Line Search, $c_1$ | 1.e-4 | 1.e-4 | 1.e-4 | - |
| Line Search, $\rho$ | 0.8 | 0.8 | 0.8 | - |
| tolerance | 1.e-14 | 1.e-14 | 1.e-14 | 1.e-14 |

We will now examine the performance of the optimisers as we vary the batch size. Firstly, we will take a look at the training accuracy.

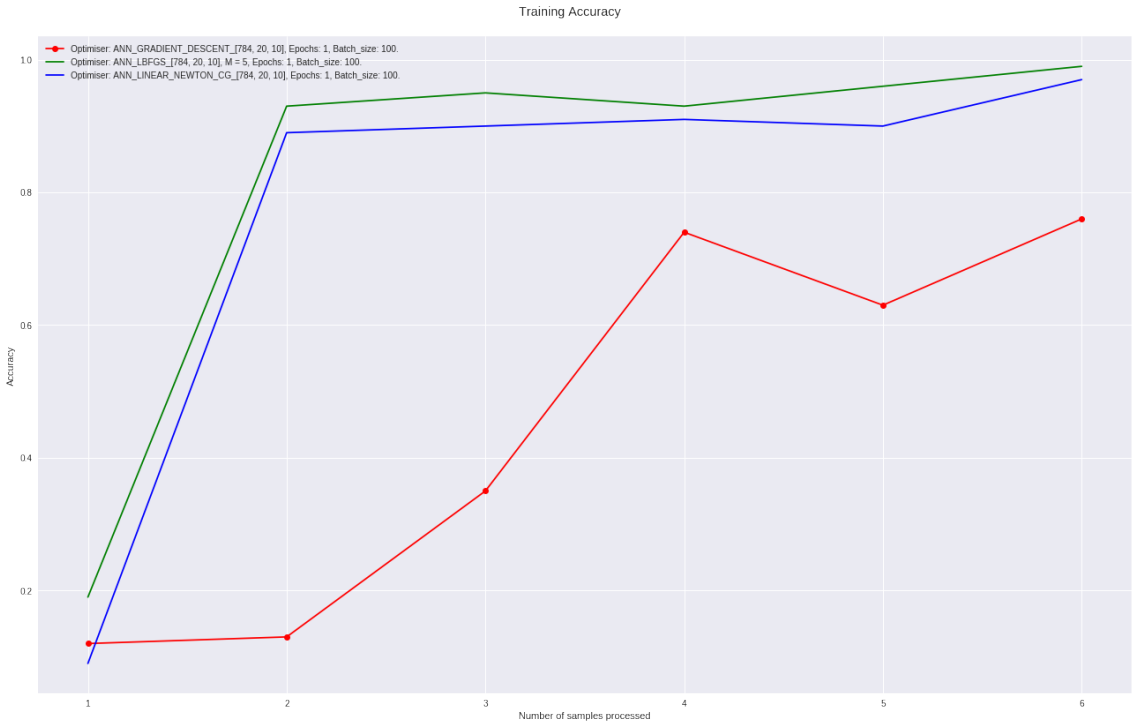


Figure 28: 100 Sample Batch Neural Network Training Accuracy.

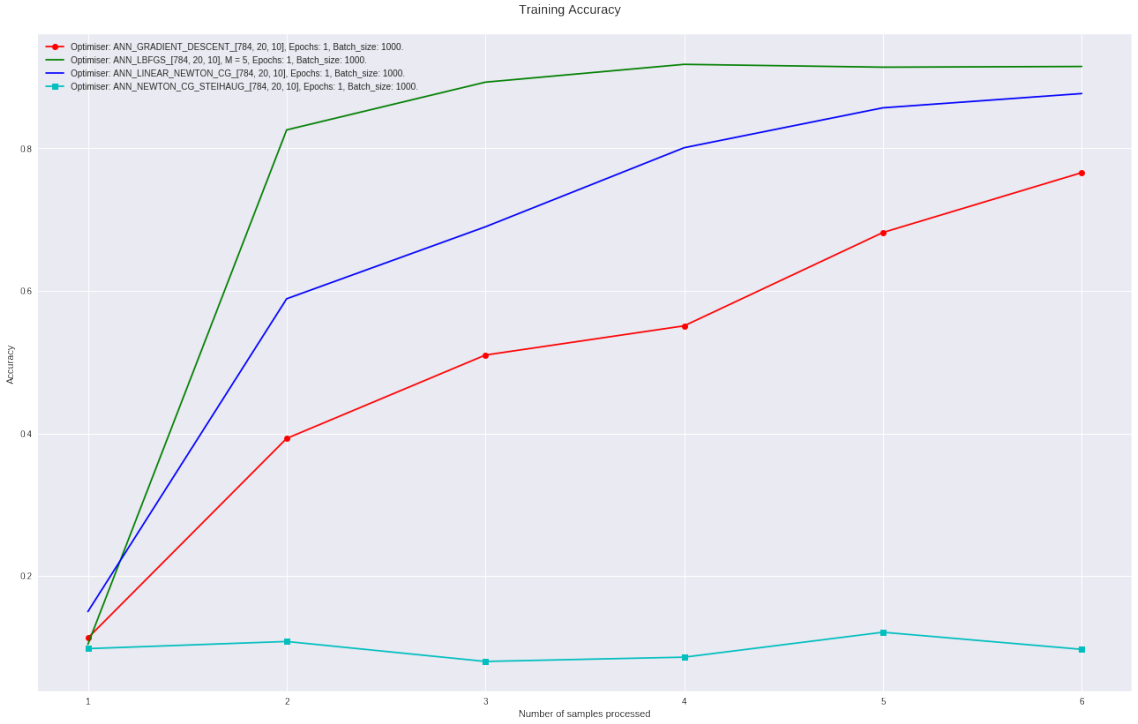


Figure 29: 1000 Sample Batch Neural Network Training Accuracy.

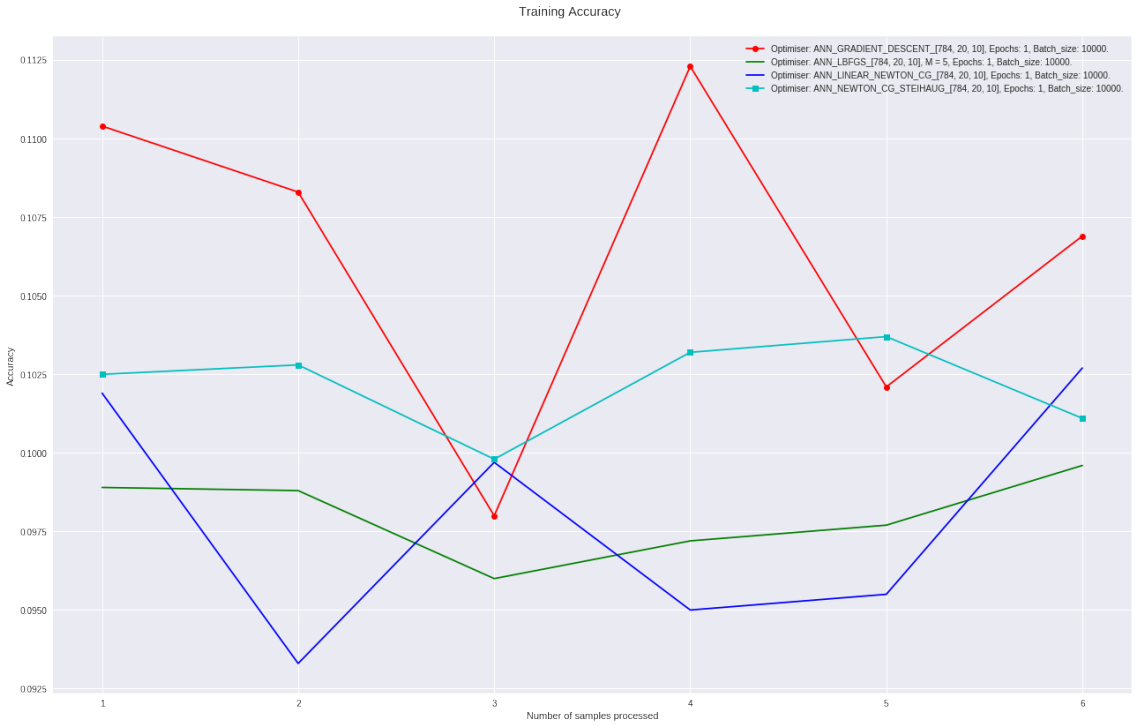


Figure 30: 10000 Sample Batch Neural Network Training Accuracy.

Like before, the LBFGS and Newton Conjugate Gradient methods tend to perform much better than the Gradient descent, both achieving higher training accuracies than the gradient descent method over a single epoch. However, for the 10000 sample batch case, it seems that over a single epoch, none of the optimisers are effectively learning anything from the optimisation problem.

Moving onto the loss plots to further with our idea that LBFGS is a much more robust method, we can empirically see that LBFGS with this setting reduces the objective function more quickly than any other method; with Newton Conjugate Gradient close behind.


Figure 31: 100 Sample Batch Neural Network Loss.

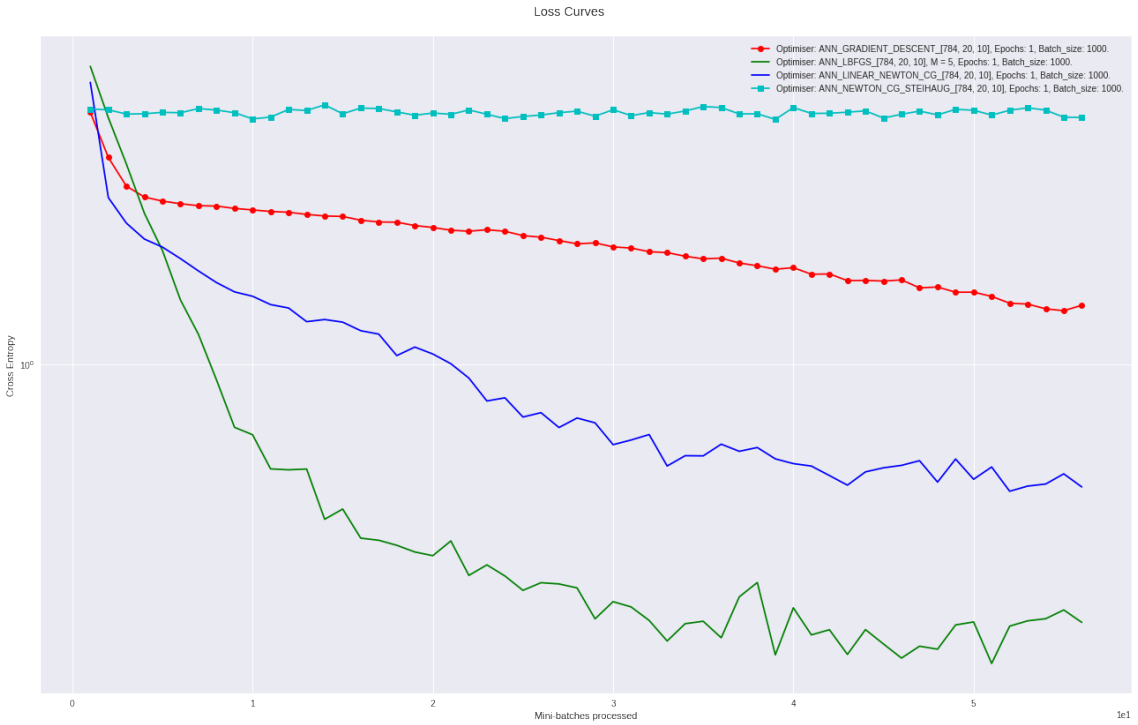


Figure 32: 1000 Sample Batch Neural Network Loss.

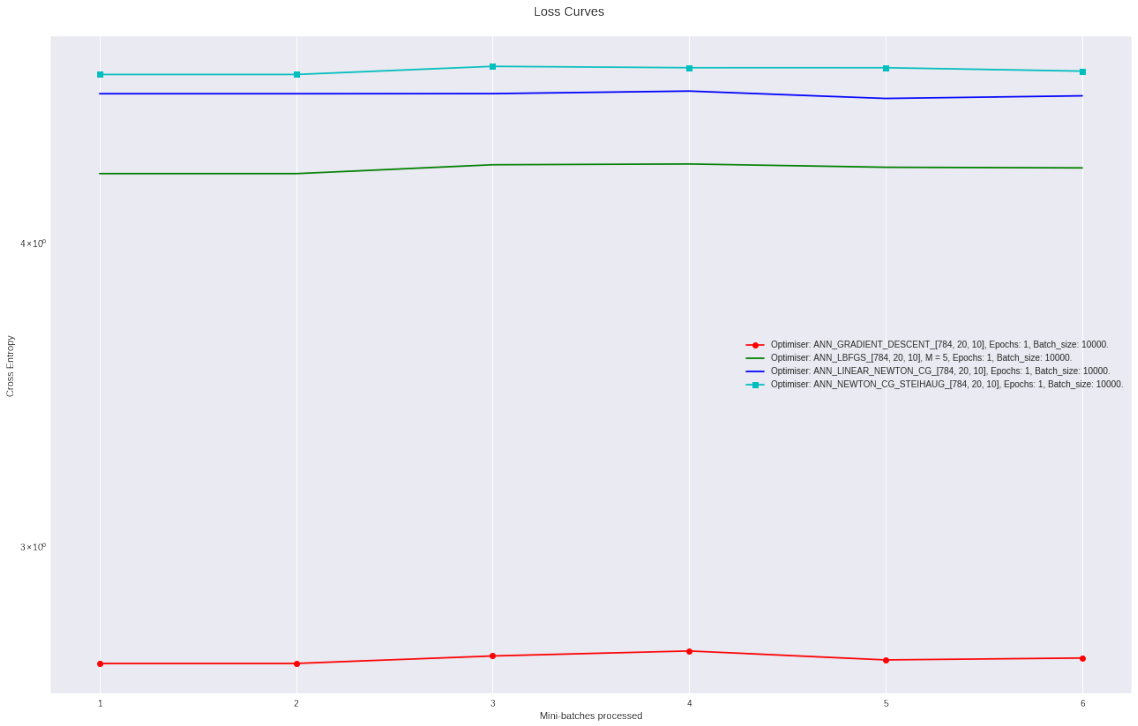


Figure 33: 10000 Sample Batch Neural Network Loss.

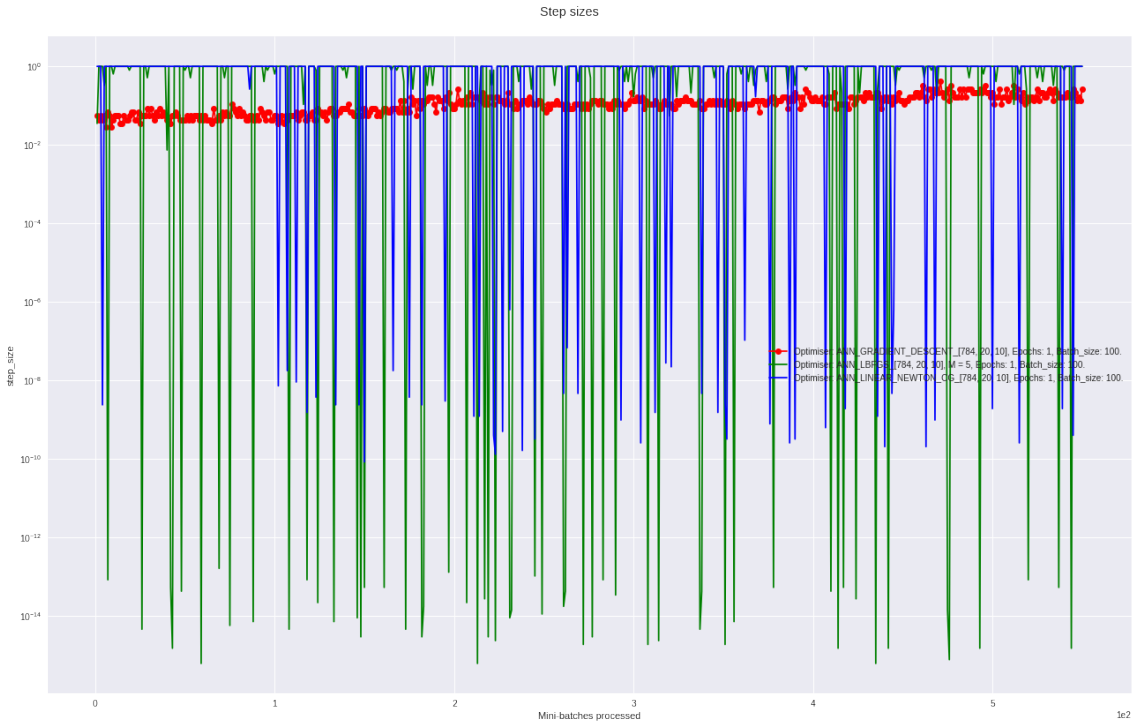


Figure 34: 100 Sample BatchNeural Network Step sizes.

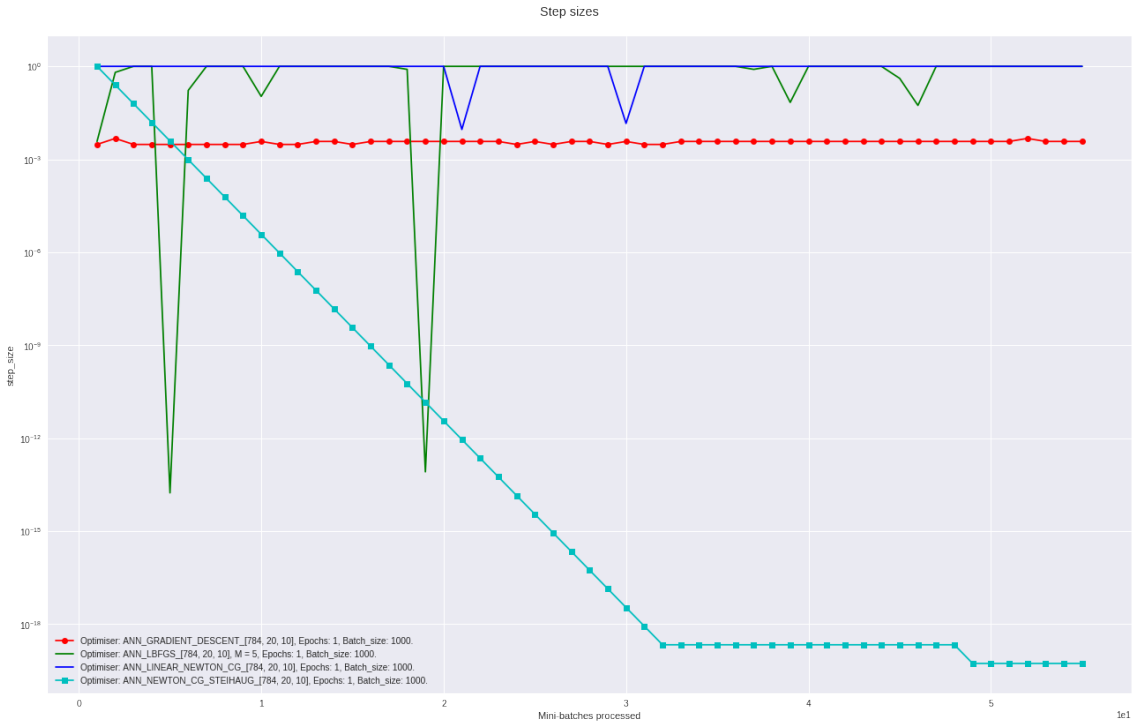


Figure 35: 1000 Sample Batch Network Step sizes.

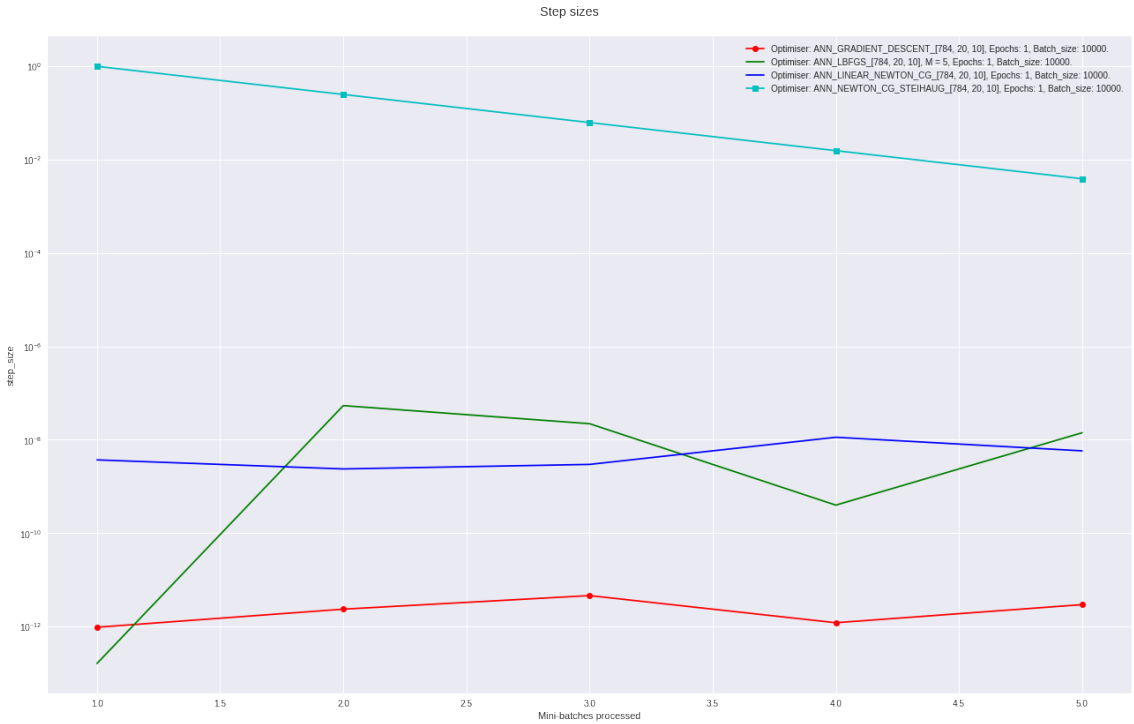


Figure 36: 10000 Sample Batch Neural Network Step sizes.

In conclusion, it would seem that the LBFGS method with a specifically chosen window, $m$, is the most robust method in minimising a loss function in terms of mini-batches processed regardless of mini-batch size and depth of neural network. However, this may not be the case when we consider actual computation time. As we had mentioned in the algorithm description section, the complexity of the LBFGS is potentially worse than any of the other methods due to the choice of $m$, the window size.

# References

[LeC] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[mni] https://knowm.org/wp-content/uploads/screen-shot-2015-08-14-at-2.44.57-pm.png.