Numerical Optimisation: Mini Project

Student Number:13064947, John Goodacre

April 2018

1 Preamble

First of all I apologise for the length of this mini-project. The project proposal proved to be akin to a wish list, with many implementations. In addition, because of my belief that the proposal should only contain 'the question' and not the mathematics, I have also added the mathematical formulation of the question in this document.

Included in the project submission are three jupyter notebooks. Each has been pre-run, with all code and charts for this document pre-generated. A word of warning - some of the neural networks (particularly the convolutional neural network) take some time to train.

Everything has been written from scratch, neural nets, and optimisers, no machine learning libraries have been used whatsoever. It adds to the length and pain of implementation, but also rewards understanding.

There was to be an additional section at the end about a recent research paper [6], moving from optimisation , to generalisation. (The crux being there are generalisation benefits to training fast in stochastic gradient descent as well as the obvious time effects). However, I have taken pity on the reader/ marker due to being highly conscious the length of this assignment has completely overstayed its welcome.

2 Introduction

The mini-project comprises an implementation and exploration of various gradient descent algorithms applied to a well known machine learning task, classification of the MNIST dataset, created by Yann Le Cunn [8]. The data itself comprises hard-written digits from 0-9 (50,000 for training purposes, together with another 10,000 unseen test images). Each image is of dimension 24x24, thus every classification function I implement in this project is a mapping from the 784 dimensional input images to a probability distribution over the 10 classes.

For prediction purposes the argmax over this output distribution is selected.

MNIST is really used due it being a known test-bed for machine learning purposes, it is well known enough to be able to calibrate results and large enough to be a serious task. The core of the project is not about MNIST though, it is really about the implementations. Several classification functions and a variety of optimisation algorithms were implemented from scratch.

In machine learning there are libraries, such as Tensorflow, Keras and Pytorch. These are heavily optimised and particularly useful for the neural networks I shall be implementing. All of these libraries, do a lot of the heavy lifting with respect to both neural net creation, the error function, calculating error gradients and the optimiser itself.

In this project I therefore decided that no libraries should be allowed whatsoever. **Philosophically, I wish to go from maths to code and algorithms should be coded from a clean sheet of paper (or screen)**. Python was used for coding (the MNIST data was pulled in from Tensorflow for convenience, but no use of Tensorflow libraries was actually made. Numpy was used simply for handing vector and matrix multiplications...of which there were a lot!).

Thus, the neural networks, the error function, all gradients, and the optimisers are coded without libraries. Due to everything being written from scratch, the code is not optimised and for some architectures and tasks (in particular convolutional neural networks), the computational load is heavy and training takes some time (up to an hour on a quad-core 3GHz machine).

2.1 Deliverables and Tests

I will flesh out the problem in greater mathematical detail later. However, suffice to say that for stochastic gradient descent and particularly for non-convex error surfaces, one quickly moves away from the standard and known convergence bounds available elsewhere in the optimisation field. I will of course give results where known, but much of the experiments will maintain an empirical flavour.

Key deliverables will be exploring stability, epoch (or sample) efficiency, as well as training time efficiency against loss rates and accuracies. This will be across architectures, but also in the second part of the project with a fixed architecture across different optimisation algorithms. The list of implementations is given below. A descent high level overview of many of the algorithms can be found in Ruder, [13].

2.1.1 Implementations from zero - no libraries:

- **Architectures implemented**
  - Linear plus Softmax - (L2 regularisation optional)
  - Multi-Layer Perceptron - (Relu activations)
  - Convolutional Neural Net
- **Optimisers implemented**
  - Batch/Stochastic/ Gradient Descent - (Fixed step/ Munro Robbins)
  - Batch/Stoch/ Grad Descent - Momentum
  - Batch/Stoch/ Grad Descent - Nesterov
  - Batch/Stoch/ Grad Descent - Adam
  - Batch/Stoch/ Grad Descent - Nadam
  - Batch/Stoch/ Grad Descent - AMSGrad (the new Adam fix)
  - Batch/Stoch/ Grad Descent - Nesterov AMSGrad (my try at a recipe!)
  - Batch/Stoch/ Grad Descent - Hypergradient Descent
- **Tests**
  - Architectures/ Batches/ L2
    - MultiClass Logistic Regression - Demo predictions
    - Gradient Descent v Batch - Simple Model results
    - Variance of Stochastic Gradient Descent
    - Munro-Robbins variable learning rate
    - L2 regularisation - Norm of weights, and training effect
    - Convolutional Neural Network - Train/ Test results
    - Multi-Layer Perceptron - Train/ test results
  - MLP - Batch / SGD
    - Vanilla v Momentum v Nesterov
    - Adam v Nadam
    - Adam v AMSGrad v NamsGrad
    - Hyper-gradient descent
    - The best

3 Classification function, Errors and Gradients

The ingredients for everything in this project comprise some form of parameterised classification function, an error (given training examples, how 'far' are our classification function outputs from our training target labels) and some gradients (how does the error change if we change our parameters?). This makes it helpful to choose a differentiable error function. Finally by using an optimiser we change these parameters to improve our function (improve meaning to reduce the loss of the error function by adjusting our parameters). In this section I will stick to functions, errors and gradients.

3.1 Error

I misunderstood the project proposal to mean that we asked the question in the proposal and were not supposed to put the maths in at that stage. I have therefore placed a mathematical formulation here.

The output from each classification function I create is going to be a matrix of dimension (batch-size x 10). Where the input data X is some batch of dimension (batch-size x 784). In short the function will produce a 10 dimensional row vector of probabilities of belonging to each class for every image in the batch. This output will be of the same form regardless of whether I use a multi-class logistic regression, multi-layer perceptron or convolutional neural net (sure the function composition gets a bit more involved in some cases, but the form of the output is the same).

For the training set, we have targets to compare our predictions against. Given we have a (1x10) vector of probabilities for each image we one-hot encode the targets. This is cute because now we can write a likelihood function given a target vector t_n. This has been seen many times before so I will be terse (also b can be subsumed into W with inputs of ones so I can reduce notational load). **T** is an (NxK) matrix of target variables.

$$p(\mathbf{T}|X, W) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k|X_n, W)^{t_n,k} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{n,k}^{t_n,k}$$

So the point is if we take the negative logarithm we have (for a set of target inputs X):

$$E(W) = -\ln p(\mathbf{T}|W) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{n,k} \ln y_{n,k}$$

In python I just take my input X's, target y's, use the model to output the probabilities of class membership for the whole dataset and compute the negative log-likelihood (I take the mean because I want to compare losses for different batch sizes, it is immaterial to this part of the discussion).

Listing 1: Python Loss function

```
def loss(self, X, y_true):
    probs = self.probs(X)
    return -(y_true * np.log(probs)).mean()
```

3.2 Error Gradient

To do gradient descent we need gradients. I apologise for interchanging between some old notation I have used and the current (I seem to use z or a for activations without prior warning).

Let $L = NLL(t, y)$, where given k classes $L = -\sum_k t_k log y_k$.

Here

$$y_k = \frac{e^{z_k}}{\sum_m e^{z_m}}$$

Clearly this is an individual loss, for the total loss we will sum over all the individual losses for the mini-batch. Taking the derivative of the softmax, we will have two cases depending upon whether $i = k$ or not.

$$\frac{\partial y_k}{\partial z_i} = \frac{\sum_m e^{z_m} e^{z_i} - e^{z_k} e^{z_i}}{(\sum_m e^{z_m})^2} \text{ when } i = k$$

$$\frac{\partial y_k}{\partial z_i} = \frac{-e^{z_k} e^{z_i}}{(\sum_m e^{z_m})^2}$$

Thus

$$\frac{\partial y_k}{\partial z_i} = y_i(1 - y_k), \text{ if } i = k, \text{ and } \frac{\partial y_k}{\partial z_i} = -y_i y_k \text{ otherwise}$$

We want

$$\frac{\partial L}{\partial z_i} = -\sum_k t_k \frac{\partial}{\partial z_i} \log y_k, \text{ so } \frac{\partial L}{\partial z_i} = -\sum_k \frac{t_k}{y_k} \frac{\partial y_k}{\partial z_i}$$

We can split the above mentioned two cases into one equation, giving

$$\frac{\partial L}{\partial z_i} = \sum_{k \neq i} \frac{t_k}{y_k} y_i y_k - t_i(1 - y_i t_i)$$

Therefore

$$\frac{\partial L}{\partial z_i} = \sum_k t_k y_i - t_i = y_i - t_i$$

Or in vector form,

$$\frac{\partial L}{\partial z_i} = y - t$$

For example ,in the case of say MNIST this would be a 10 dimensional object (where errors are added for mini-batches). Note that we can always use this loss. When we move to neural networks we backprop from this loss in the output layer (reverse auto-diff). Which is really all an application of the chain rule where one needs to keep careful track of dimensions.

3.2.1 Loss with respect to the input x

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial x_j}$$

but given that $z_i = \sum_j W_{i,j}x_j + b_j$ we have that

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial z_i} W_{i,j}$$

Therefore in vector form $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W = (y-t)W$

3.2.2 Loss with respect to the weights W

$$\frac{\partial L}{\partial W_{j,k}} = \sum_i \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial W_{j,k}}$$

Therefore

$$\frac{\partial L}{\partial W_{j,k}} = \sum_i \frac{\partial L}{\partial z_i} x_k \delta_{ij} = \frac{\partial L}{\partial z_j} x_k$$

In vector form

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial z}\right)^T x^T = (y-t)^T x^T$$

3.2.3 Loss with respect to the biases b

$$\frac{\partial z_i}{\partial b_j} = \delta_{ij}, \text{ therefore } \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} = y - t$$

It's simpler in code...

Listing 2: Python Simplest Gradients - Linear Softmax

```
def calc_gradients(self, X,y):
    train_probs = self.probs(X)

    grad_weights = X.T@(train_probs - y)
    grad_biases = np.sum(train_probs - y)
```

Again note that if we do stochastic gradient descent over a mini-batch we would sum over all the mini-batch samples.

3.3 Linear plus Softmax

3.3.1 Function Output

$$p(C_k|x) = y_k(x) = \frac{e^{a_k}}{\sum_j e^{a_j}} \text{ , where } a_k = (x^T W + b)_k$$

So in our case for a single data item x, the probability it belongs to class k, is given by the softmax value of the k'th column of an affine transformation. To be absolutely clear on dimensionality, each input from MNIST is a 28x28 image (represented as a 784 dimension column vector). We turn this into a row vector (1x784). Multiply by our weights and add our biases (784x10 + 1x10), which gives a (1x10) output of probabilities, the kth element of this is our $p(C_k|x)$. Clearly this can all be transposed other ways (depending on how we present our matrix of weights and our inputs) and as long we grab the correct outputs!

This is for a single datum x, if we expand to the whole dataset X. Then X is of dimension (50,000 x 784), so our output probabilities will be of dimension (50,000 x 10). This we do for full gradient descent, for batch stochastic gradient descent we will just have (batch-size x 10) as our set of output probabilities and stochastic will be a lonely output vector of dimension (1 x 10).

Bar the function output, clearly the loss and gradients have been derived above for this case, so we are done.

3.3.2 L2 Regularisation

This is going to be a big mini-project to I will only give a 'helicopter view'. A couple of ideas here, which I don't have space to expand upon, but just to show I am aware. First what is it? ...in the briefest made up notation.

$$Lreg(:,:,:) = L(:,:,:) + \lambda ||W||_2^2$$

Here with my bad notation, I have a loss function. I regularise it. I do so using a 2-norm. Why? First it can make non-convex things convex (I imagine adding positives to diagonals of a matrix until it is positive semi definite - I don't have better analogies in my head right now). Second it penalises large norms for W. Third - it is properly differentiable (thats handy of course, but doesn't drop features like L1, but c'est la vie).

Why is this useful? Well with neural networks we have many parameters and can overfit, this is one of the means of reducing tortuously high parameter values which may actually only be fitting noise in the training data (not learning some real underlying manifold of the data). For a simple model such as a Softmax on top of an affine transformation regularisation may not be so useful. I suspect it lacks power and so all the regulariser will really do is constrain it. But these are the ideas anyway, I mention it because I have coded it, but then the project grew and I moved away from it. So just for completeness.

3.4 Multi-layer Perceptron

The multi-layer perceptron I implemented had the following architecture. An initial layer of 128 neurons, a second layer of 64 neurons and a final layer outputting the usual affine transformation plus a softmax. Thus there were (784x128+128) weights and biases in the first layer. (128x64+64) weights and biases in the second layer and (64x10+10) weights and biases in the final layer. After each affine transformation, I passed the outputs through a relu non-linearity. The function and its derivatives are one liners in python.

Listing 3: Python Relu and its derivative

```
def relu(x): return np.maximum(x, 0)
def relu_prime(x): return np.greater(x, 0).astype(int)
```

Clearly this network is bigger and more powerful as a classifier than the first. The only difference as regards training the network is the need to calculate gradients for all the extra parameters. This was done through backpropagation. We simply need the partial derivatives of the error with respect to all the weights and biases. The code snippet is given below.

Listing 4: Python Backprop MLP

```
def activations(self ,X):
    """Given input data X, weights and biases and a set of
    functions to apply non-linearities returns a list of
    all activations...the final layer will be logits"""

    a = [X]

    for w,b,f in zip(self.weights, self.biases, self.functions):
        a.append(f[0](a[-1].dot(w) + b))
    return a


def calc_gradients(self, X,y):
    """Calculates all gradients and biases for the neural network
    with a forward and backward pass"""

    a = self.activations(X)

    gradw = np.empty_like(self.weights)
    grad_bias = np.empty_like(self.biases)

    #Output layer errors
    delta = (a[-1] - y)
    gradw[-1] = a[-2].T.dot(delta)
    grad_bias[-1] = np.sum(delta, axis=0)

    #Backprop the errors through the rest of the network
    for i in range(len(a)-2,0,-1):

        #self.functions[i-1][1] is the not the function itself,
        #but its derivative now...
        delta = self.functions[i-1][1](a[i])*delta.dot(self.weights[i].T)
        grad_bias[i-1] = np.sum(delta, axis=0)
        gradw[i-1] = a[i-1].T.dot(delta)
    grad_bias = grad_bias / len(X)
    gradw = gradw / len(X)

    #Store the last data - helpful for some later optimisation algos
    self.last_X = X
    self.last_y = y
    return gradw, grad_bias
```

3.5 Convolutional Neural Net

The final classification architecture I implemented was a convolutional neural network. The structure comprised an input layer (this time batch-size x 28 x 28). We don't flatten the input images now. Then a convolution layer of (3x3x8), i.e. eight 3x3 convolutions. Then a max-pooling layer (2x2). The convolution and max pooling layers were repeated before being output into a fully connected network (relu activations) and our softmax classifier. The 3x3 convolutions are passed over all patches of each image, and the max pooling gathers the strongest activations.

The idea is to take advantage of the natural hierarchical structure of many images. Although there are less parameters than a fully connected network (as we have weight sharing and disparate pixels are not connected in the network), there is unfortunately a heavy computational load in calculating all convolutions in all the images.

The trained weights will be precisely those kernels that best aid in classifying the images. In contrast to other areas of machine vision where predefined kernels may be used (such as Sobel edge detectors). Here the advantage is that the network can itself learn the best kernels.

The convolutional neural network was the one area where all the code is not my own. I again coded everything from scratch and didn't use any libraries. However, my initial implementation was too slow. The key is vectorisation, but even so I found I needed to use some helper functions provided in a course by Stanford University (basically all the im2col stuff and some of the forward/ backward pass stuff is sped up using their functions). Even so the CNN despite its power takes a long time to train and when I did later work on the optimisers I tended to concentrate on the MLP architecture in order to get more results.

3.5.1 Convolutional Neural Net gradients

Because we are going to train the network using gradient descent (or its variants), we again need to calculate the gradients of the weights in the CNN. This is more complex than it looks (the answer is basically a transposed convolution), but because we have a lot of dimensions, the notation can get a little hairy. My attempt at derivation is given below.

3.5.2 Derivative of convolutional layer w.r.t parameters W and input x, filter size H x W x D, stride 1

I will start by defining some terms. I am only showing the derivative with respect to the convolution layer. Thus the input x is post any prior activation function or pooling. I assume the output y from the convolution layer is prior to the next module which would likely be another non-linear activation function followed by pooling.

I am using L as my loss and presuming that through back-prop we will already know the derivative of the loss with respect to the output Y of our convolution layer. From this point I will calculate the derivative with respect to the weights W and input X.

In terms of coordinates - I will use i,j to find my way around the kernel (one channel to begin with), the output coordinates I will use i', j'. So super-script dashes refer to outputs. The height and width of the kernel are H,W (not to be mistaken with my choice of W for the weights).

3.5.3 One Channel

One channel - $\frac{\partial L}{\partial W}$

For a forward pass through the convolution layer we thus have:

$$Y_{i',j'} = \sum_{i}^{H} \sum_{j}^{W} W_{i,j} X_{i'+i-1,j'+j-1} + b$$

Taking the derivative with respect to W

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{i'}^{H}\sum_{j'}^{W} \frac{\partial L}{\partial Y_{i',j'}} \frac{\partial Y_{i',j'}}{\partial W_{i,j}} = \sum_{i'}^{H}\sum_{j'}^{W} \frac{\partial L}{\partial Y_{i',j'}} X_{i'+i-1,j'+j-1}$$

and we know $\frac{\partial L}{\partial Y_{i',j'}}$ through backprop to the output of our convolution layer. Note that this itself is another convolution.

3.5.4 Multiple Channels - adding another dimension

Keeping the same notational format, let d be the number of channels in the previous layer and d' the number of layers in the next. This is similar to the prior reasoning but there are a few more sub-scripts to keep track of.

$$Y_{i',j',d'} = \sum_{i}^{H}\sum_{j}^{W}\sum_{d}^{D} W_{i,j,d,d'} X_{i'+i-1,j'+j-1,d} + b_{d'}$$

and again taking derivatives with respect to the weights $\frac{\partial L}{\partial W}$.

$$\frac{\partial L}{\partial W_{i,j,d,d'}} = \sum_{i',j',d'} \frac{\partial L}{\partial Y_{i',j',d'}} \frac{\partial Y_{i',j',d'}}{\partial W_{i,j,d,d'}}$$

Therefore,

$$\frac{\partial L}{\partial W_{i,j,d,d'}} = \sum_{i',j',d'} \frac{\partial L}{\partial Y_{i',j',d'}} X_{i'+i-1,j'+j-1,d}$$

And as previously we again have a convolution.

If we now take derivatives with respect to the input $\frac{\partial L}{\partial X}$

$$\frac{\partial L}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial L}{\partial Y_{i',j',d'}} \frac{\partial Y_{i',j',d'}}{\partial X_{i,j,d}}$$

therefore we have that (apologies for more notation adding e's, but I hope this is clear)

$$\frac{\partial L}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial L}{\partial Y_{i',j',d'}} \frac{\partial}{\partial X_{i,j,d}} \left(\sum_{m,n,e} W_{m,n,e,d} X_{i'+m-1,j'+n-1,e} + b_e \right)$$

So if we again take only non-zero derivatives i.e. where $i = i' + m - 1$, $j = j' + n - 1$, then we have $m = i - i' + 1$ and $n = j - j' + 1$. Giving us finally that:

$$\frac{\partial L}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial L}{\partial Y_{i',j',d'}} W_{i-i'+1,j-j'+1,d,d'}$$

For a mini-batch we will pass through another dimension forwards and have a batch of errors which are summed and passed back. The short story for implementation purposes is that we are passing convolutions forwards and another transposed convolution backward.

The full implementation can be found in the file 'Fromzero-Conv.ipynb' attached to this assignment. However, for practical purposes although this all works, this is the stage at which the speed benefits of using libraries such as tensorflow come into their own. My own implementation takes an hour to train.

4 Optimisers

Please note that some of this exposition and the notation is a combination of my own writing and notation from Sebastian Ruder, who desribes many of these algorithms very cleanly [13].

4.1 Theoretical Results

As I mentioned in the project proposal, in my research searching for convergence guarantees I found that smooth is good, convex is good, strongly-convex even better. For example there are various asymptotic and known bounds in cases of convexity, particularly the case of strong convexity, for example, Rakhlin et al. 'Making Gradient Descent Optimal for Strongly Convex Stochastic Optimization', [11]

In my chosen area of exploration, optimisation applied to current large scale non-convex representations such as neural networks [4]. I did not really find theoretical guarantees. Several of the algorithms have asymptotic guarantees in the convex case. Indeed Bottou has written papers on the tricks of the trade [3] for stochastic gradient descent applied to neural networks, so the practioner world appears to be driven by heuristics.

Because I wished to explore some sizeable implementations using non-convex neural networks, I'm afraid therefore the project took more of an empirical turn, with results being experiments between optimisers (and architectures). I tried to do this in some detail, however it does mean that the more satisfying experiments where one can relate back to theoretical optimisation bounds I certainly couldn't find, and possibly do not yet exist for non-convex neural networks.

4.2 Full Batch, Mini-Batch, and Stochastic Gradient Descent

After the fairly heavy duty overhead of creating our classification function, our error function and all the gradients. The gradient descent algorithm is very simple, particularly compared to some of the more complex algorithms presented on the course, it is about as simple as it gets. Take a step in the direction of steepest descent of the error function with respect to the a parameters. And repeat. In the course we covered many more sophisticated methods, particularly second order methods where the Hessian or its approximations are calculated. This works very well in convex cases and can of course work with non-convex error surfaces. However, there is often also an overhead for large high-dimensional data sets as if common with neural networks.

The three variants I implemented were full gradient descent (calculate the gradient using the whole training set, before taking a step), stochastic (grab one random training example and take a step based on this example - which itself could be a step in the wrong direction, but works in expectation), or grab a random mini-batch (I choose 200) and calculate the gradient through each mini-batch, take a step and continue through all the training examples to comprise one epoch. There are obvious differences.

4.2.1 Batch Gradient Descent

Given an error function E, weights W and a learning rate μ. We calculate the gradient of our cost function with respect to the weights for the whole training set, then take a step.

$$W_{t+1} = W_t - \mu \nabla_{W_t} E(W_t)$$

In practice this has disadvantages which I will demonstrate in my experiments. The most obvious is that this quickly becomes intractable for massive data-sets, the less obvious is that this is inefficient in data samples, we need to calculate the gradient after processing all 50,000 training examples in the case of MNIST, before even taking a single step. This is unnecessary.

4.2.2 Stochastic Gradient Descent

This is the other extreme, we perform a parameter update for every training example x_n and its label y_n. Our update is thus:

$$W_{t+1} = W_t - \mu \nabla_{W_t} E(W_t; x_n; y_n)$$

There are some convergence results, which I describe above. In practice, stochastic gradient descent is far more data efficient as regards steps than batch gradient descent. The trade off being variance. There is no guarantee that the gradient of the error as regards a single training example is actually an overall descent direction. The clever part of the algorithm is that we are travelling in a descent direction in expectation. However, the learning rate is important and indeed it should either be annealed for convergence or awareness that a fixed learning rate will cause 'jumps' around a local minimum (neural nets are non-convex so we may not find global minima).

4.2.3 Mini-batch Stochastic Gradient Descent

As one would expect this is a combination between the two. We randomise the training set, choose a batch size, calculate our error and gradients and update. Thus we have multiple updates for each epoch, but with greater stability than the single training datum stochastic gradient approach.

$$W_{t+1} = W_t - \mu \nabla_{W_t} E \left(W_t; x_{[n:n+batch-size]}; y_{[n:n+batch-size]} \right)$$

The code for the above variants of gradient descent is as simple as the maths.

Listing 5: Python Vanilla Gradient Descent

```
wt_step = learning_rate * grad_weights
bias_step = learning_rate * grad_biases

def train_model(self, wt_step, bias_step):
    self.weights -= wt_step
    self.biases -= bias_step
```

4.3 Momentum

Qian et al describe momentum and convergence properties in [10]. The algorithm effectively helps stochastic gradient descent by pushing it along the direction of a valley when plain vanilla gradient decent might oscillate back and forth.

$$\nu_t = \gamma \nu_{t-1} + \mu \nabla_{W_t} E(W_t)$$

$$W_{t+1} = W_t - \nu_t$$

4.4 Nesterov

Nesterov introduced what I consider a rather clever and non-intuitive optimiser in, 'A method for of solving a convex programming problem with convergence rate $O(1k^2)$' [9]. The lookahead mechanism effectively approximates the gradient not with respect to the current weights but the approximate future position of the weights.

Note that convergence guarantees appear to be only for the convex case. Not for neural networks. However, this is still widely used. Indeed a recent paper by Botev 'Nesterov's accelerated gradient and momentum as approximations to regularised update descent', places Nesterov in a wider context [2]. The update methematically and code implementation is given below.

$$\nu_t = \gamma \nu_{t-1} + \mu \nabla_{W_t} E(W - \gamma \nu_{t-1})$$

$$W = W - \nu_t$$

The code listing is given below, and is a little more involved as there is a second set of weights calculated whose gradients are then calculated before taking the descent step.

Listing 6: Python Nesterov

```
elif self.opt_type == "Nesterov":

    self.nest_wt = np.array(self.weights) - self.alpha * self.v1
    self.nest_bias = np.array(self.biases) - self.alpha * self.v2

    nest_gradwt, nest_gradbias =
        self.nest_calc_gradients(self.last_X, self.last_y)

    self.v1 = self.v1*self.alpha + learning_rate * nest_gradwt
    self.v2 = self.v2*self.alpha + learning_rate * nest_gradbias

    wt_step = self.v1
    bias_step = self.v2
    return wt_step, bias_step
```

4.5 Adam

ADAM was created in the paper 'Adam: A method for stochastic optimization', by Kingma and Diederik [7]. The authors offered a convergence guarantee in the case of convex functions. At the time of the project proposal. I found that the proof of this guarantee is now being questioned. A paper at the time under double blind review (and now accepted) for potential presentation at ICML 2018, 'On the convergence of Adam and beyond', [12], suggests that ADAM has no convergence guarantees even for convex functions. Adam and its fix was potentially to be implemented in the bonus section of the project proposal. However, I found neither too onerous and both are included in this project.

The idea is to compute averages which anneal the past gradients and the past gradients squared.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{W_t} E(W_t)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_{W_t} E(W_t))^2$$

Because m_t and v_t are biased, these biases are then corrected.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Finally the weights are updated:

$$W_{t+1} = W_t - \frac{\nu}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

The code snippet for Adam is given below, for brevity I have only shown the weights update, not biases:

Listing 7: Python Adam

```
elif (self.opt_type == "Adam"):
    self.Adam_m_1 = self.Adam_Beta_1*self.Adam_m_1 +
            (1-self.Adam_Beta_1) * np.array(grad_weights)

    self.Adam_v_1 = self.Adam_Beta_2*self.Adam_v_1 +
            (1-self.Adam_Beta_2) * np.array(grad_weights)**2

    Adam_m_1_hat = self.Adam_m_1/(1-self.Adam_Beta_1)
    Adam_v_1_hat = self.Adam_v_1/(1-self.Adam_Beta_2)

    sqrt_Adam_v_1_hat = np.empty_like(Adam_v_1_hat)
    sqrt_Adam_v_2_hat = np.empty_like(Adam_v_2_hat)

    #Necessary as I have a list of weights for the layers
    #of the neural net
    for i in range(Adam_v_1_hat.shape[0]):
        sqrt_Adam_v_1_hat[i] = np.sqrt(Adam_v_1_hat[i])
        sqrt_Adam_v_2_hat[i] = np.sqrt(Adam_v_2_hat[i])

    wt_step = (learning_rate/(sqrt_Adam_v_1_hat + self.Adam_e))
                                                * Adam_m_1_hat
    return wt_step
```

4.6 AMSGrad

The recent paper by Reddi et al. [12], shows that there are potential problems with the convergence of Adam. The crux of the matter is the use of the exponential average to update the paramters v_t. The authors even demonstrate a simple convex problem where Adam fails to converge. I was going to replicate this example, but others have already done so, for example https://fdlm.github.io/post/amsgrad/ where a number of tests are implemented using Theano, and also the original authors have posted the convex example using Tensorflow online.

I therefore decided to continue the theme of the project and simply implement AMSGrad from scratch also. This sounds bigger than it is and in fact the adjustment is very minor. And the similarities with Adam above are very obvious.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

This is the big change !

$$\hat{v}_t = max(\hat{v}_{t-1}, v_t)$$

$$W_{t+1} = W_t - \frac{\mu}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

The introduction of the max of the past squared gradients changes the step size update and solves the above mentioned convergence problem.

4.7 Nadam

Nadam was not mentioned in the project proposal in either the standard or bonus sections. But given that I implemented both Adam and Nesterov, it seemed reasonable to also implement Nadam. Nadam comes from Nesterov-accelerated Adaptive Moment Estimation and is described in Dozat 2016, [5]. The idea is of course to incorporate nesterov acceleration into Adam. I have given the final update equation below, the implementation is in the Jupyter notebook.

$$W_{t+1} = W_t - \frac{\nu}{\sqrt{\hat{v}_t} + \epsilon} + (\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t})$$

4.8 NAMSGrad

Again this algorithm won't be found in the project proposal or the bonus as I basically just made it up! This is just my own implementation of AMSGrad but using Nesterov updates. It is basically the AMSGrad version of Nadam. I have no idea if it is any good or its convergence properties as I basically made it up for this project. The code listing is given below.

Listing 8: Made up algorithm NAMSGrad

```
#This is my attempted implementation of the new AMSGrad
#with Nesterov updates

elif self.opt_type =="NAMSGrad":

    self.Adam_m_1 = self.Adam_Beta_1*self.Adam_m_1 +
                    (1-self.Adam_Beta_1) * np.array(grad_weights)
    self.Adam_m_2 = self.Adam_Beta_1*self.Adam_m_2 +
                    (1-self.Adam_Beta_1) * np.array(grad_biases)

    self.Adam_v_1 = self.Adam_Beta_2*self.Adam_v_1 +
                    (1-self.Adam_Beta_2) * np.array(grad_weights)**2
    self.Adam_v_2 = self.Adam_Beta_2*self.Adam_v_2 +
                    (1-self.Adam_Beta_2) * np.array(grad_biases)**2

    Adam_m_1_hat = self.Adam_m_1/(1-self.Adam_Beta_1)
    Adam_m_2_hat = self.Adam_m_2/(1-self.Adam_Beta_1)

    Adam_v_1_hat = self.Adam_v_1/(1-self.Adam_Beta_2)
    Adam_v_2_hat = self.Adam_v_2/(1-self.Adam_Beta_2)

    for i in range(Adam_v_1_hat.shape[0]):
        self.Adam_v_1_hat[i] = np.maximum(self.Adam_v_1_hat[i],
                                          Adam_v_1_hat[i])
        self.Adam_v_2_hat[i] = np.maximum(self.Adam_v_2_hat[i],
                                              Adam_v_2_hat[i])

    sqrt_Adam_v_1_hat = np.empty_like(Adam_v_1_hat)
    sqrt_Adam_v_2_hat = np.empty_like(Adam_v_2_hat)

    for i in range(Adam_v_1_hat.shape[0]):
        sqrt_Adam_v_1_hat[i] = np.sqrt(self.Adam_v_1_hat[i])
        sqrt_Adam_v_2_hat[i] = np.sqrt(self.Adam_v_2_hat[i])


    wt_step = (learning_rate/(sqrt_Adam_v_1_hat + self.Adam_e)) \
    * (self.Adam_Beta_1 * Adam_m_1_hat + (((1-self.Adam_Beta_1) \
    *np.array(grad_weights))/(1 - self.Adam_Beta_1)))

    bias_step = (learning_rate/(sqrt_Adam_v_2_hat + self.Adam_e)) \
    * (self.Adam_Beta_2 * Adam_m_2_hat + (((1-self.Adam_Beta_2) \
    *np.array(grad_biases))/(1 - self.Adam_Beta_2)))
    return wt_step, bias_step
```

4.9 Hypergradient Descent

This has been a long journey of both maths and code implementations, however I had also mentioned in my project proposal another bonus section where given time (and energy!). I would explore hypergradient descent. The motivation for this came from research on ADAM and adaptive gradients and indeed the paper, 'The marginal value of adaptive gradient methods in machine learning' Wilson et al [14].

I became interested in a paper by Baydin et al. [1] 'Online Learning Rate Adaptation with Hypergradient Descent'. This algorithm implements stochastic gradient descent, but at each step implements gradient descent on the step size itself. I also found what I felt to be a good MSc Thesis by David Rubio, from Oxford University. (Convergence Analysis of an adaptive method of gradient descent, 2017).

Hypergradient descent is the application of gradient descent on the learning rate of an underlying gradient descent algorithm. It uses the partial derivative of the objective function, after an update step with respect to the learning rate.

Following the notation and borrowing some equations from the original paper [1]. We have our regular gradient descent algorithm, with α being the learning rate.

$$\theta_t = \theta_{t-1} - \alpha \nabla f(\theta_{t-1})$$

We want to also have an update rule for α itself. The authors achieve this by calculating $\frac{\partial f(\theta_{t-1})}{\partial \alpha}$.

$$\frac{\partial f(\theta_{t-1})}{\partial \alpha} = \nabla f(\theta_{t-1}).\frac{\partial(\theta_{t-2} - \alpha \nabla f(\theta_{t-2}))}{\partial \alpha} = \nabla f(\theta_{t-1}).(-\nabla f(\theta_{t-2}))$$

Therefore the update rule is...

$$\alpha_t = \alpha_{t-1} + \beta \nabla f(\theta_{t-1}).\nabla f(\theta_{t-2})$$

and

$$\theta_t = \theta_{t-1} - \alpha_t \nabla f(\theta_{t-1})$$

So the the overhead is basically a matrix multiplication and keeping an extra copy of the gradients. The paper shows the case for a scalar sequence of α's. The MSc thesis goes into far more detail about the mathematics.

4.9.1 Observations

Clearly given the low extra over-head and generality of this framework, it can be used on pretty much any of the above gradient descent algorithms. I only implemented it for plain vanilla gradient descent, simply due to the fact this project is already out-sized.

On implementation I was confused about a couple of things glossed over in the paper. At the start their gradients are initialised to zero, so I couldn't see how a dot product with this start point would result in updated hypergradients being away from zero. So for my own initialisation I first performed a gradient descent step, then proceeded as per normal.

My second confusion was as regards dimensionality. With my neural networks I have a list of weight matrices of various dimensions. Taking the gradient of these weights with respect to α does not result in a scalar sequence of α's. I thus, took the simplest interpretation and simply summed all the gradients (with respect to alpha) to return a scalar. I didn't have the time but it appears to me that one could take advantage of the non-scalar nature of the differential of the weights with respect to α by having differential learning rates throughout the neural network based upon the different hypergradients. However I don't know what this does for convergence and here is probably not the place to explore.

The python code snippet of my implementation is given below.

Listing 9: Hypergradient Descent

```python
elif self.opt_type == 'Hyper_GD':
    #The most basic implementation of hypergradient descent
    #I still have a scalar sequence of learning rates.
    #This could be a vector etc.
    #I am also only allowing the learning rate to move very slightly
    #Unsure about convergence results and stability on this thing
    #It could of course be implemented for lots of more
    #sophisticated algos than SGD. But this is getting messy.

    if self.num_steps ==0:
        #Seems to me I need to do a gradient descent first step
        #else where are am I getting my first hypergradients??

        wt_step = learning_rate * grad_weights
        bias_step = learning_rate * grad_biases
        self.Hyp_learning_rate = learning_rate
    else:
        #After the first step we have a hypergradient
        #learning rate
        wt_step = self.Hyp_learning_rate * grad_weights
        bias_step = self.Hyp_learning_rate* grad_biases

    #get the next gradients
    next_grad_weights, next_grad_biases = \
        self.hyp_calc_gradients(self.last_X,
            self.last_y, self.weights-wt_step, self.biases-bias_step)

    hypergrads = []
    for i in range(len(grad_weights)):
        hypergrads.append(np.sum(-np.dot(grad_weights[i].T,
                                next_grad_weights[i])))
        hypergrads.append(np.sum(-np.dot(grad_biases[i].T,
                                next_grad_biases[i])))

    #It seems to me we are losing info just summing back to a scalar here
    #can't we be cuter than this, why not a sequence of alpha vectors??

    hypergrad = sum(hypergrads)

    #Update the hypergradient learning rate
    self.Hyp_learning_rate -= self.Hyp_Beta*hypergrad
    self.num_steps += 1

    return wt_step, bias_step
```

5 Experiments Architecture/ Loss/ Batch

5.1 Simple Softmax Model

The first experiments are pure sensibility checks to ensure the simplest model is working. The simplest model being just a multiclass logistic regression (affine transformation plus softmax).

5.1.1 Example predictions

In figure 1, I show some example predictions on some unseen test-set examples. The model was trained with batch gradient descent for 400 epochs, with a batch size of 200. This fairly weak model achieves an accuracy of around 91% on MNIST. The bottom right image is classified incorrectly.

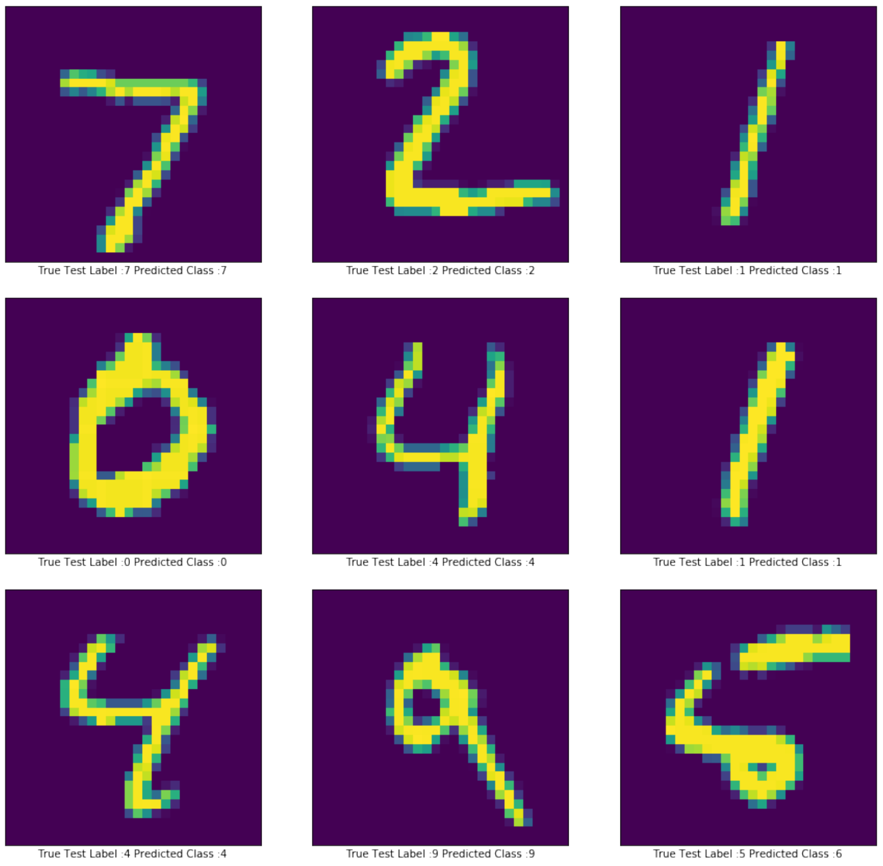

True Test Label :7 Predicted Class :7
True Test Label :2 Predicted Class :2
True Test Label :1 Predicted Class :1
True Test Label :0 Predicted Class :0
True Test Label :4 Predicted Class :4
True Test Label :1 Predicted Class :1
True Test Label :4 Predicted Class :4
True Test Label :9 Predicted Class :9
True Test Label :5 Predicted Class :6

Figure 1: Affine plus Softmax - Batch Gradient Descent - Test predictions

5.1.2 Train/ Test losses and accuracies by Epoch

Remaining with the simplest model. Figure 2, shows the train and test accuracies by Epoch for batch gradient descent versus full gradient descent. Clearly both the loss and accuracy is better by epoch using batch gradient descent, highlighting its relative efficiency with training samples. The reason of course if that many more steps are taken for each epoch (batch size was 200, and 200 epochs were run for both experiments).

5.1.3 Higher variance of stochastic gradient descent - fixed step

Figure 3, illustrates the losses and train/test accuracies for stochastic gradient descent. Each epoch comprises a single training example. The experiment comprised 5000 epochs.

5.1.4 Munro-Robbins Annealing of stochastic gradient descent

Figure 4, illustrates the losses and train/ test accuracies for stochastic gradient descent. As before the experiment comprises 5000 epochs. The difference this time being a variable learning rate to reduce the variance of the stochastic gradient descent. Munro-Robbins are convergence conditions for stochastic gradient descent (in a convex setting however), I simply chose to anneal the learning rate at $\frac{1}{epoch}$, recall that an epoch in this case is for a single training example.

5.1.5 Impact of L2 regularisation

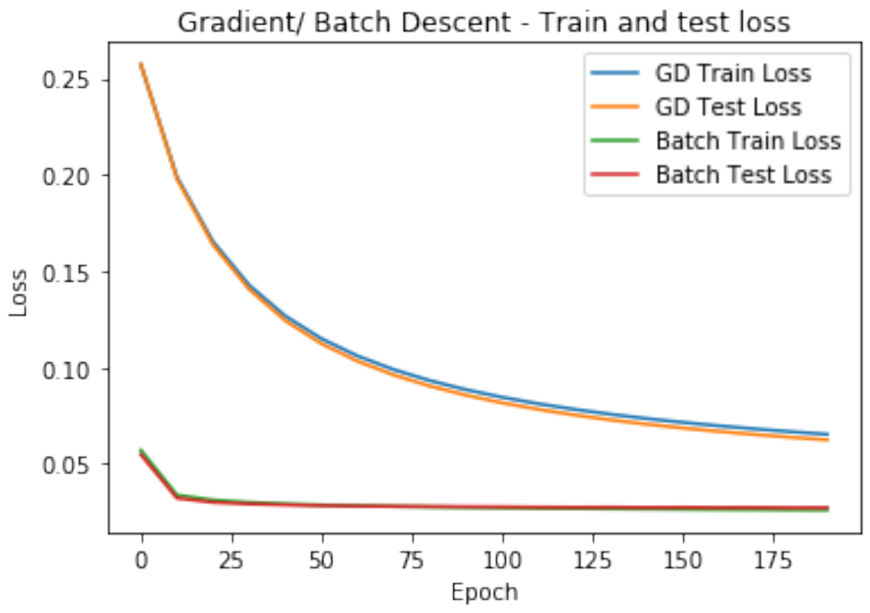
The final basic experiment with the simplest model was to illustrate the impact of L2 regularisation. I used a $\lambda = 0.001$. In figure 5, I show the norm of the weights through time as well as the train and test accuracy. The comparison is against the same model without regularisation.

As expected the norm of the weights is lower for the regularised model. It also plateaus out, with a higher loss and lower accuracy. In this case the model isn't particularly powerful, however this is useful where we have more powerful models and wish to trade off bias and variance and reduce model complexity and over-fitting.

All experiments thus far have been in the attached file 'Fromzero-OO-Copy1'. I will now move to more powerful models and tests.

5.2 Convolutional Neural Network

The most powerful model I implemented was the convolutional neural network mentioned earlier in this document. The code is in the attached document 'Fromzero-Conv.ipynb'. Figure 6 illustrates the curves of the training losses and accuracies. With only 10 Epochs, the CNN quickly reaches a training accuracy of 99%. Outstripping the other models, indeed after only 5 epochs there



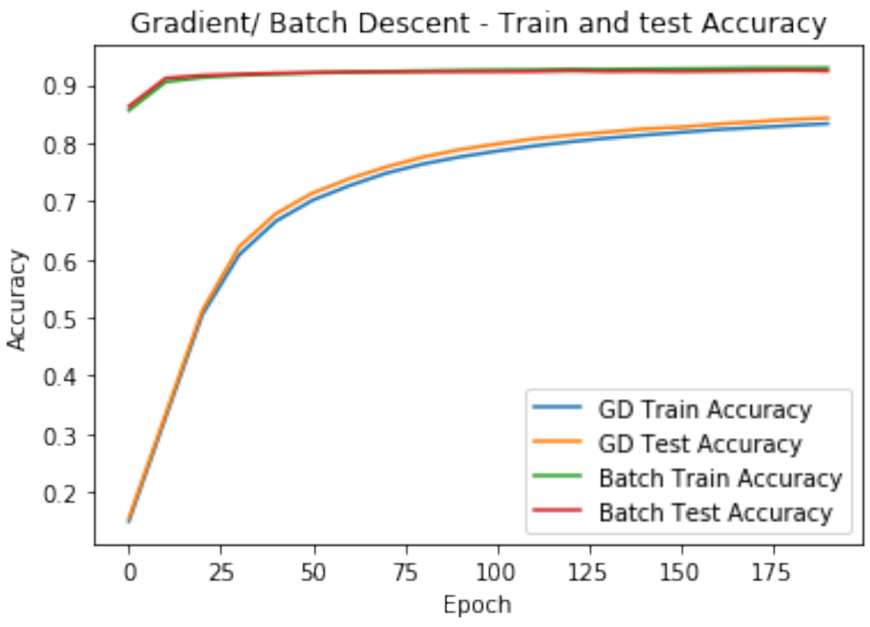
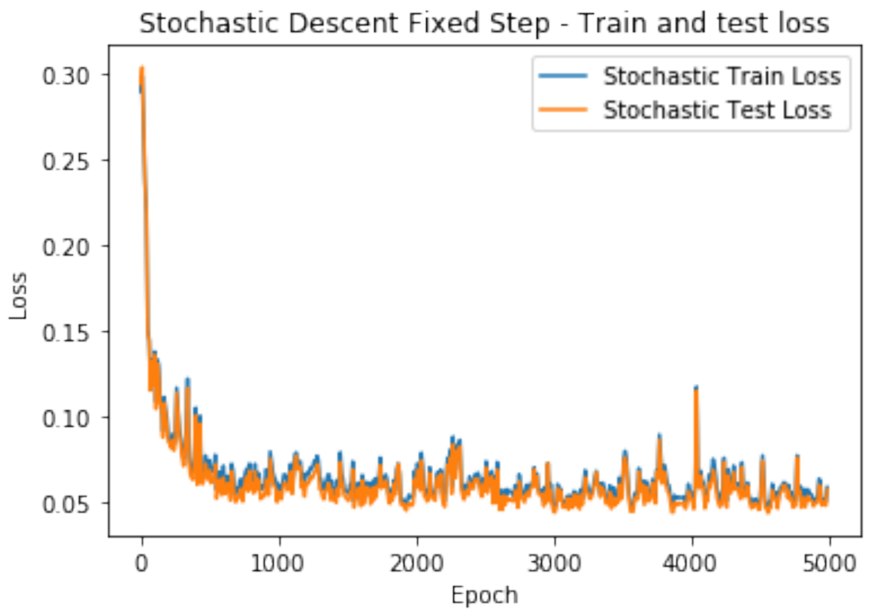


Figure 2: Data Inefficiency of Gradient Descent v Batch - Multi-Class Logistic



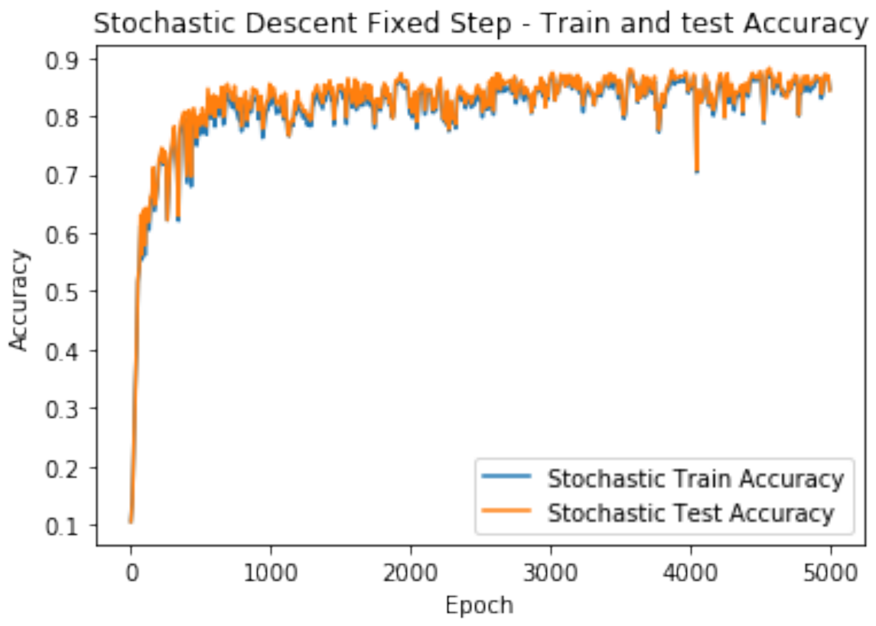
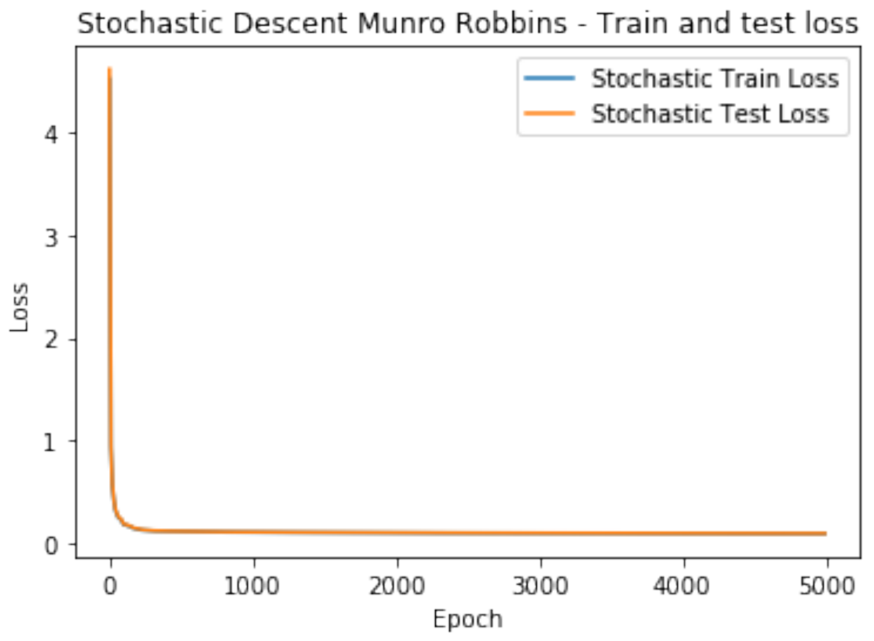


Figure 3: Higher Variance of Stochastic Gradient Descent



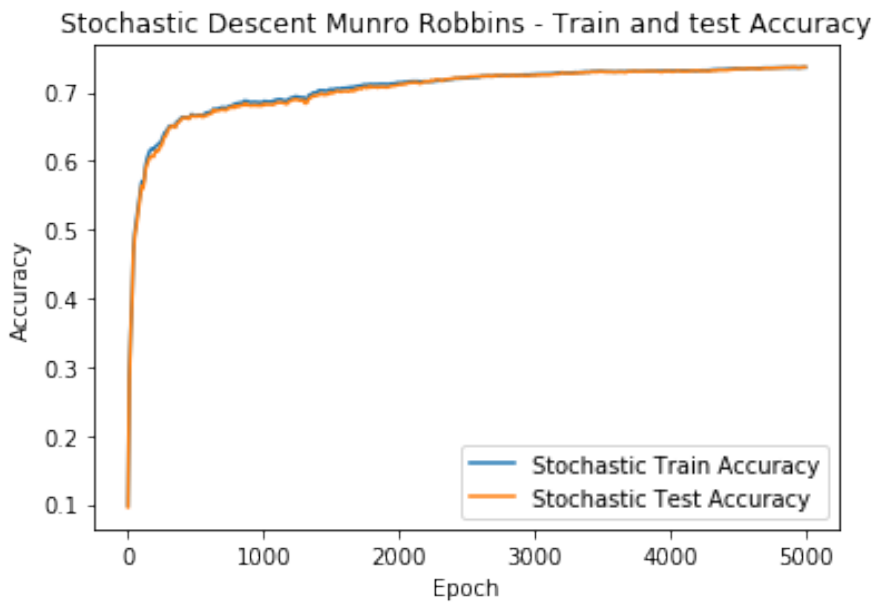


Figure 4: Lower Variance of Stochastic Gradient Descent - with Munro Robbins

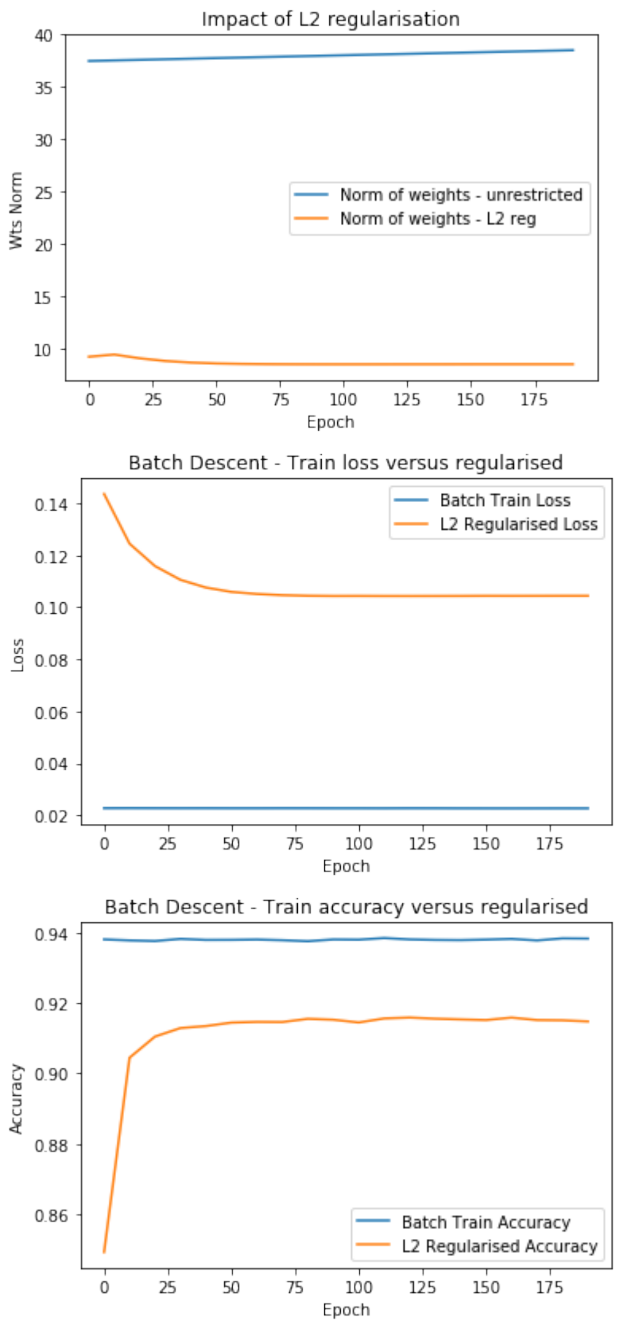


Figure 5: Impact of L2 regularisation

is evidence of over-fitting.

However for the purposes of the rest of the project the CNN was used sparingly. Although a powerful classifier, the training time is an order of magnitude greater than the other models, not particularly helpful when one wishes to run multiple test on various optimisers. Although the 'sample' efficiency was the same as for the MLP. The computational efficiency is much worse, due to the need to calculate multiple kernels across every pixel of every image.

Also it is so powerful that training is achieved with only a few epochs, making it less helpful to show relative efficiencies across optimisers, where I may wish to see more subtle gains across hundreds of training epochs.

5.3 Muli-Layer Perceptron Baseline

Figure 7, both gives a baseline of result for training the Multi-layer perceptron described earlier, as well as once again showing the sample inefficiency of full gradient descent with a different model.

Figure 8, gives a baseline for training the Multi-layer perceptron, this time with stochastic gradient descent. Once again we see the higher variance of SGD.

From here I will drop full gradient descent and concentrate on stochastic or mini-batch stochastic. I will fix upon the MLP architecture and begin comparing the different optimisers I implemented, Figure 7 simply uses plain fixed step gradient descent as a baseline.

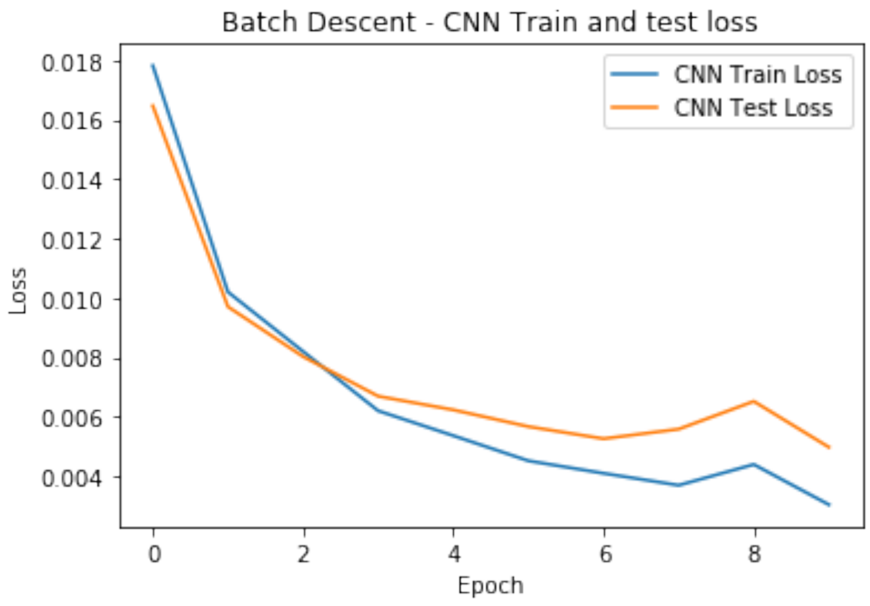
6 Optimiser Experiments

6.1 Stochastic Gradient Descent - Vanilla/ Momentum/ Nesterov

In figure 9, I illustrate the last 2000 epochs for stochastic gradient descent applied to the multi-layer perceptron. I consider the results inconclusive for this example between the three algorithms. The model is capable of far greater predictive power, given more training, but this does illustrate the volatility and number of episodes for SGD to get there, even with momentum.

6.2 Stochastic Gradient Descent - Vanilla/ Adam/ Nadam

In figure 10, I redo the last experiment but now using Adam and Nadam. Clearly both Adam and Nadam have lower losses, and greater accuracies than vanilla SGD for the same level of training.



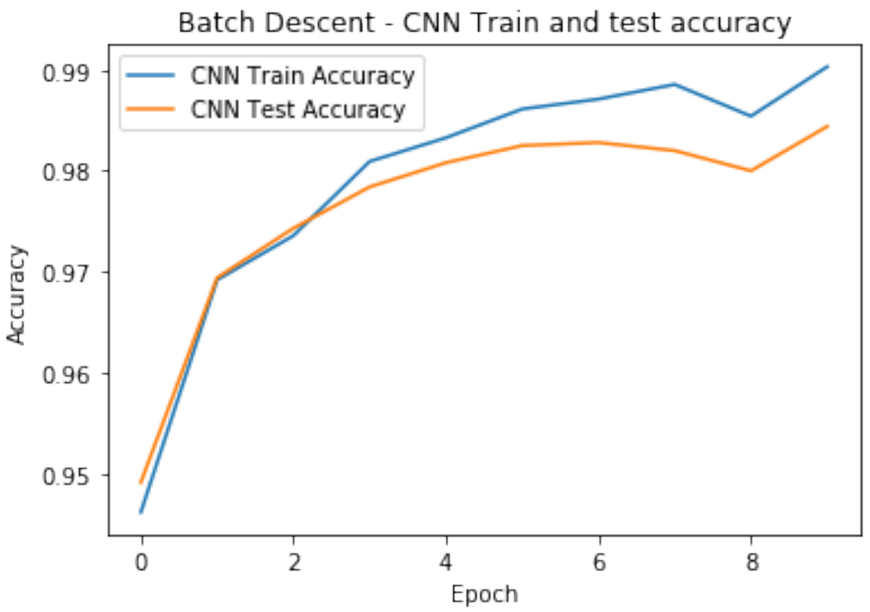
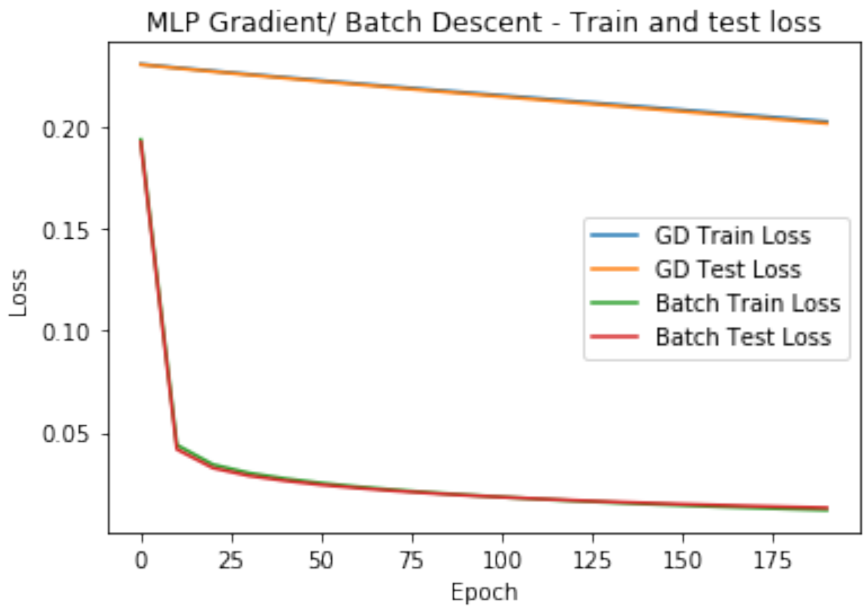


Figure 6: Convolutional Neural Net - Train/ Test loss, Accuracy



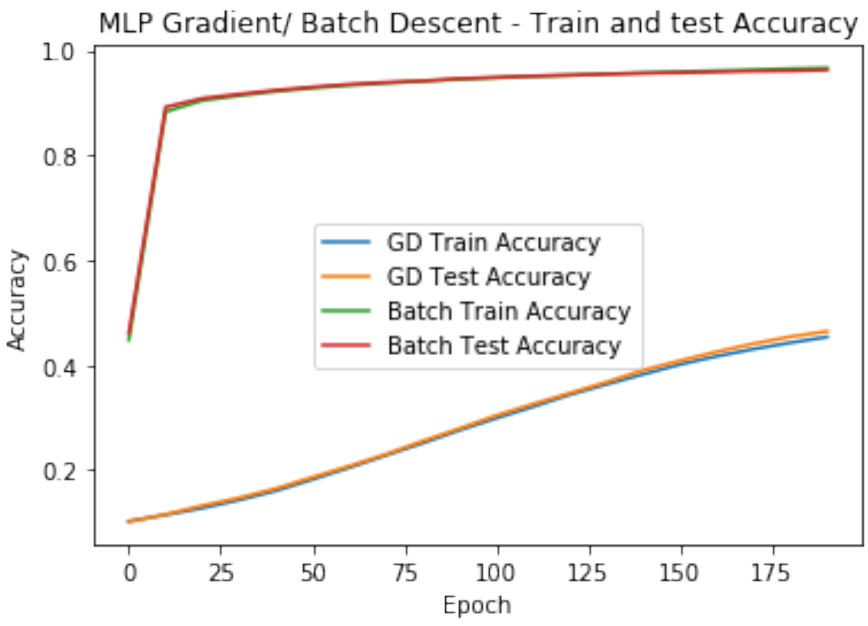
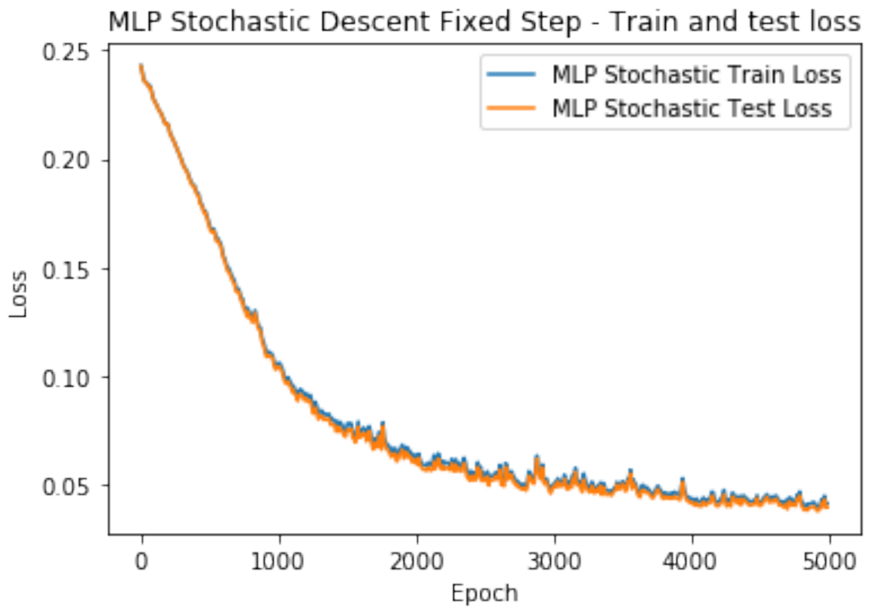


Figure 7: Multi-Layer Perceptron - Full v Mini-Batch loss, Accuracy



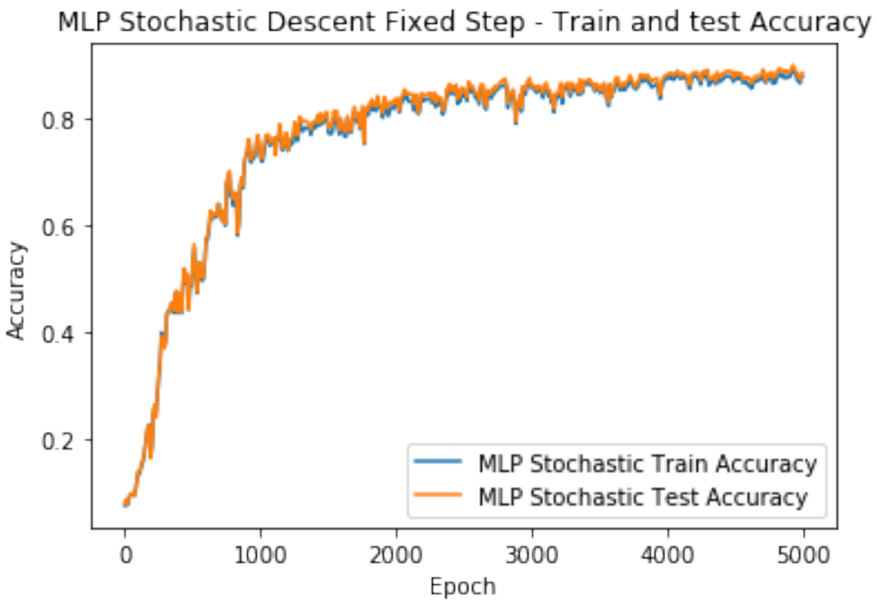
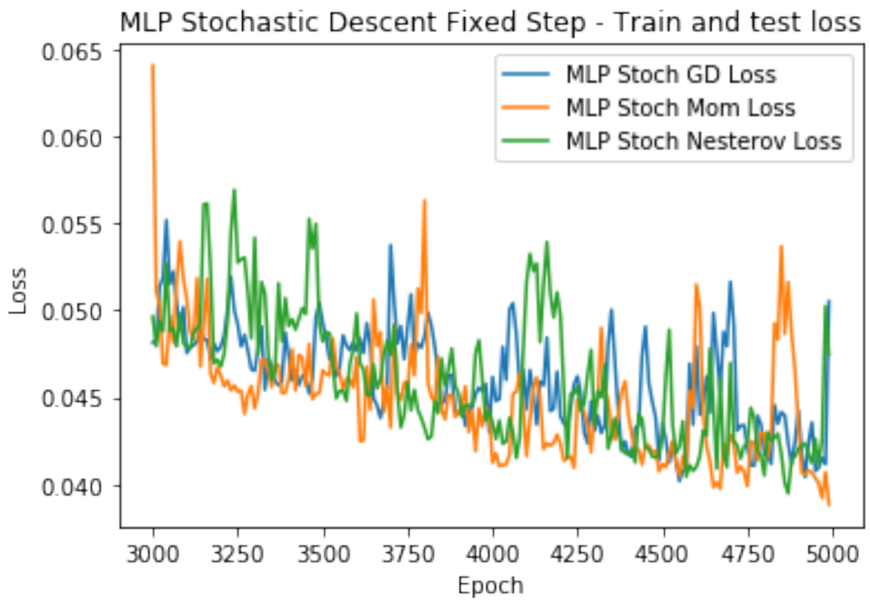


Figure 8: Multi-Layer Perceptron - Stochastic loss, Accuracy



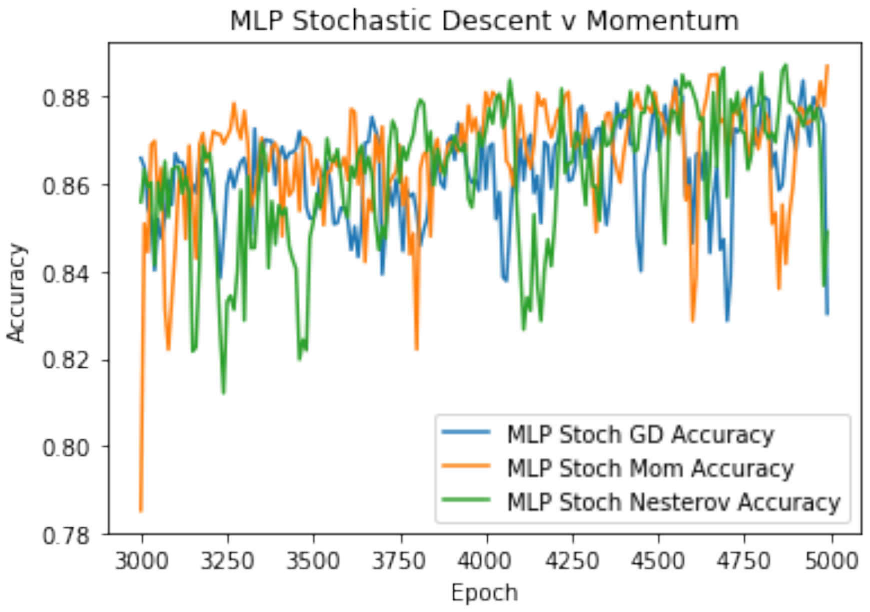
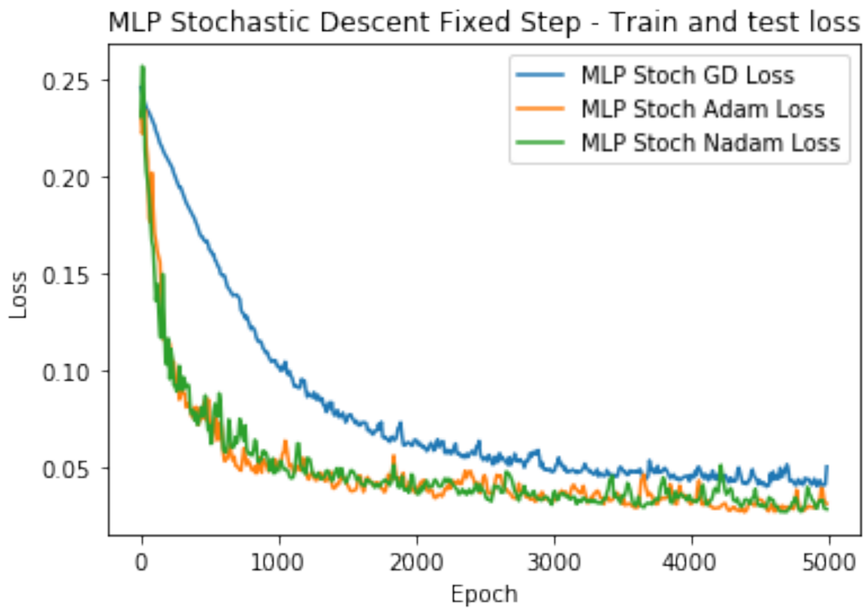


Figure 9: Multi-Layer Perceptron - SGD Vanilla/ Momentum/ Nesterov



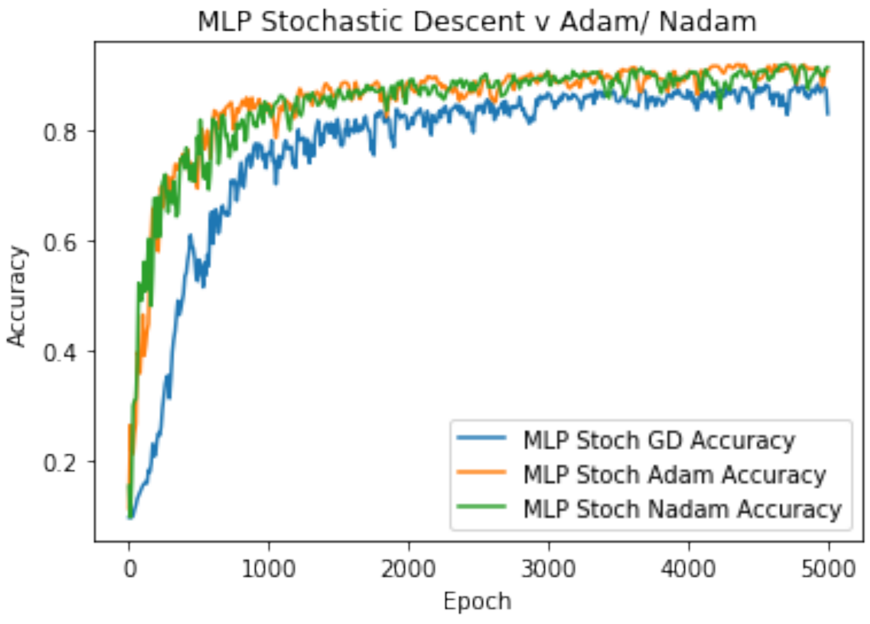


Figure 10: Multi-Layer Perceptron - SGD Vanilla/ Adam/ Nadam

6.3 Stochastic Gradient Descent - Adam/ AMSGrad/ Nams-Grad

In figure 11, I again redo the last experiment but now using Adam and AMSGrad and Namsgrad. For this experiment the results appear similar.

6.4 Stochastic Gradient Descent - Hypergradient Descent

The final experiment for vanilla stochastic gradient descent uses hypergradient descent. Figure 12 shows that hypergradient descent does appear to learn faster with lower losses than vanilla gradient descent with this model.

6.5 Optimiser Experiments - Mini-Batch SGD/ MLP

From here the experiments focus on the best set up, which is mini-batch gradient descent (I use a batch size of 200 and 200 epochs). As before I sometimes start the graphs from later epochs to more clearly illustrate differences.

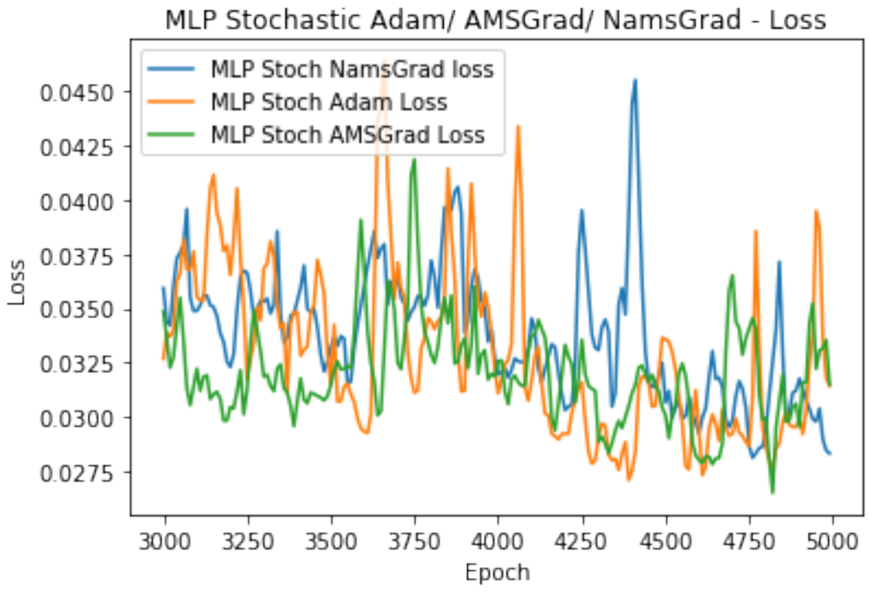
6.5.1 Mini-Batch SGD - Vanilla/ Momentum/ Nesterov

Figure 13, shows Mini-Batch gradient descent for vanilla stochastic gradient descent, momentum and nesterov momentum. The differences are that now losses are far lower and accuracies far higher. The mini-batch updates are helping the optimisers find better local minima, however for this experiment at least the three optimisers gave quite comparable results.

6.5.2 Mini-Batch SGD - Vanilla/ Adam/ Nadam/ AMSGrad/ NAMS-Grad

Figure 14, gave a few small surprises. Again we are doing mini-batch gradient descent. The model is a multi-layer perceptron. I have described the convolutional neural net as the most powerful classifier for this task. However, with only a few epochs the MLP was capable of 100% classification accuracy on the training set of 50,000 images (yes this is over-training, but I am talking about minimising a loss function, not generalisation in this task).

In short, earlier optimisers were less efficient at exploring the error surface given the same number of samples. Even a relatively small MLP is a rich enough model for this dataset. Adam, Nadam, AMSGrad and NAMSGrad were startling in their relative efficiency for this task, to the extent the graphs almost look meaningless as they go straight to 100% accuracy and zero loss. (Sensible practice for generalisation purposes would thus be using regularisation, dropout or early stopping in this case).



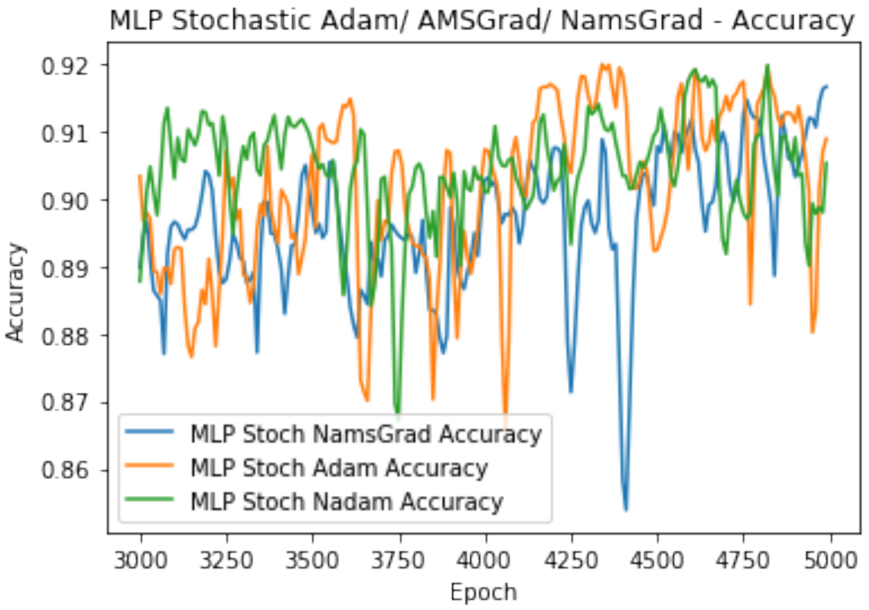
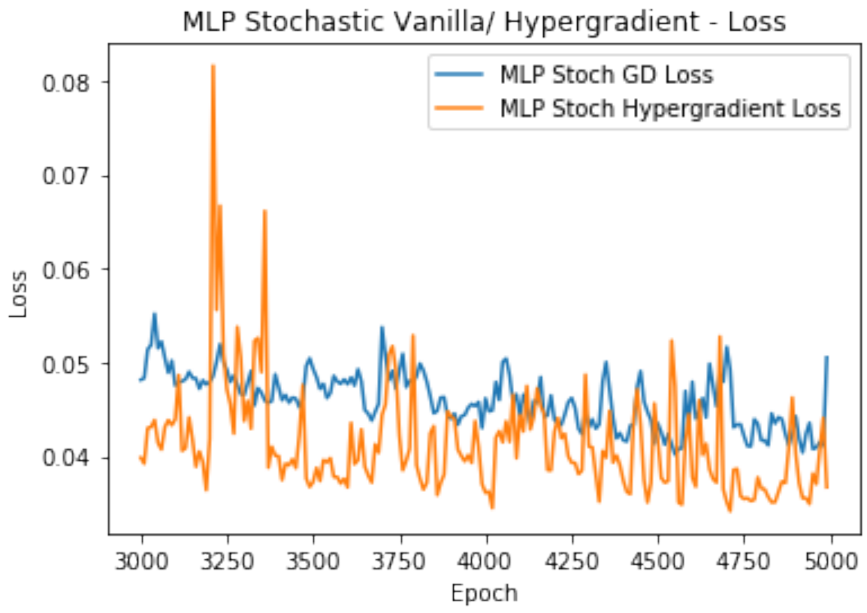


Figure 11: Multi-Layer Perceptron - SGD Adam/ AMSGrad/ NamsGrad



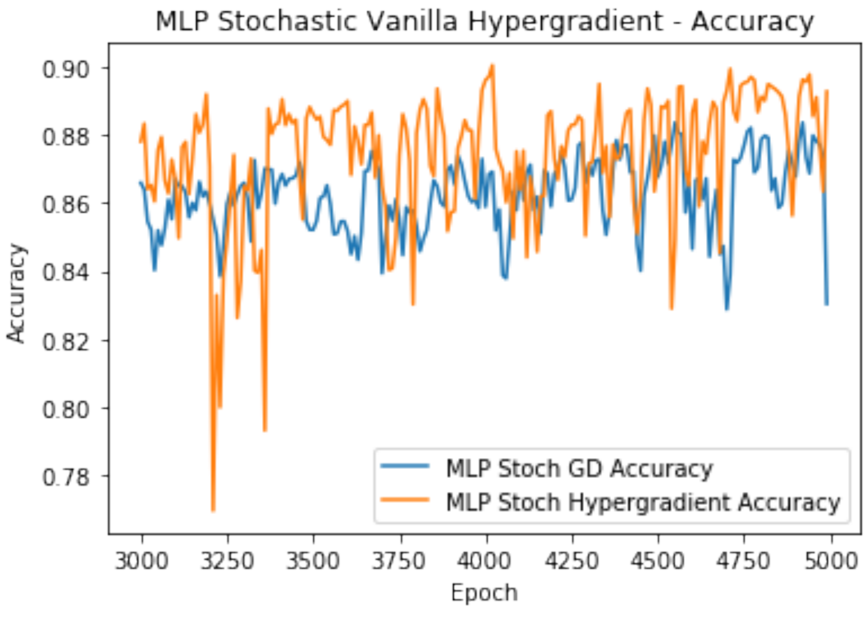
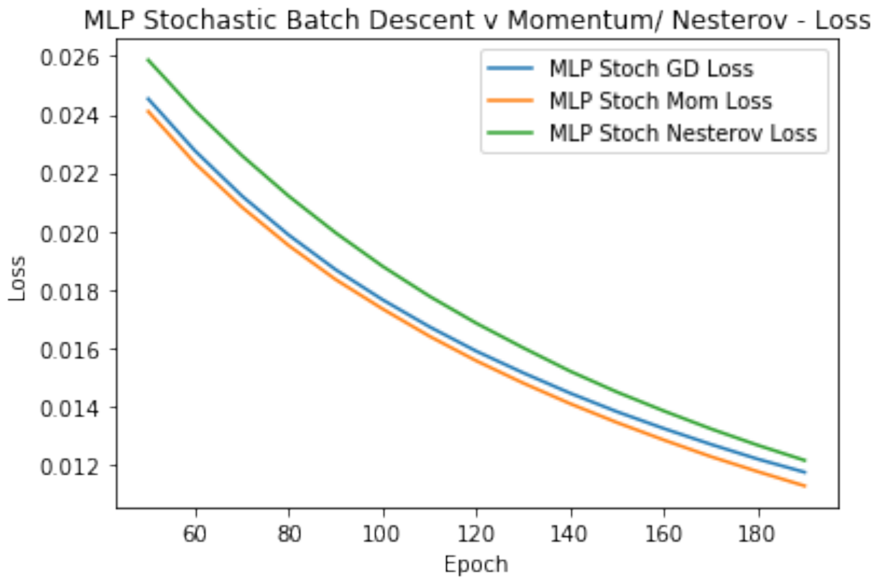


Figure 12: Multi-Layer Perceptron - SGD Hypergradient Descent



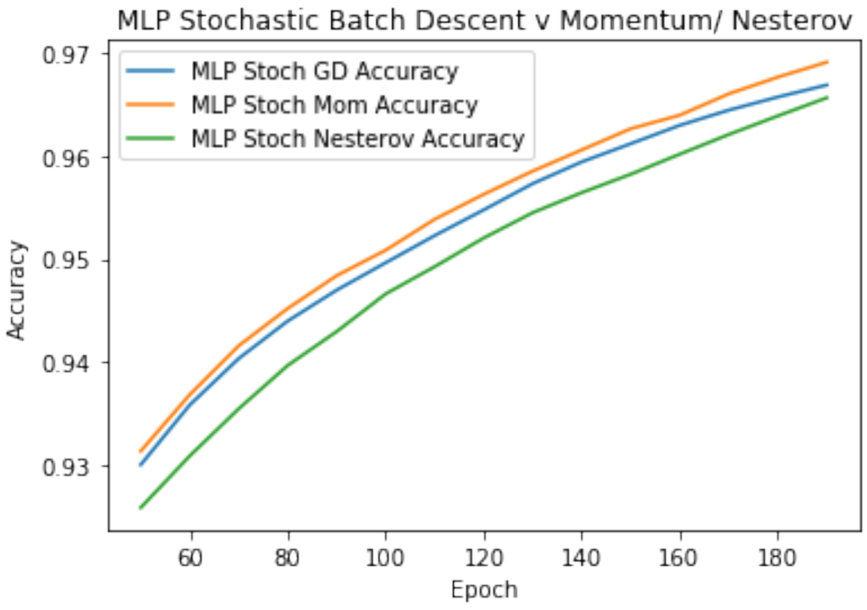
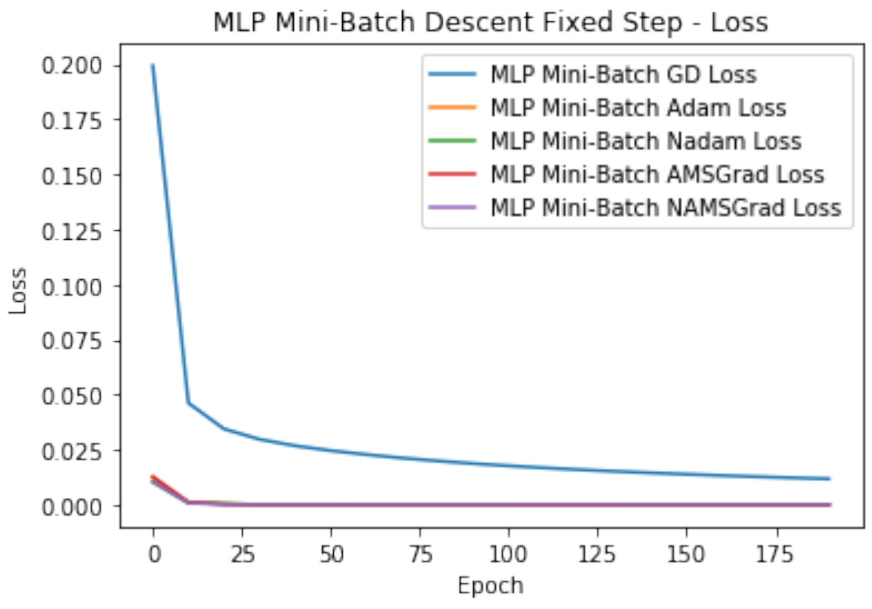


Figure 13: Multi-Layer Perceptron - Batch SGD Vanilla/ Mom/ Nesterov



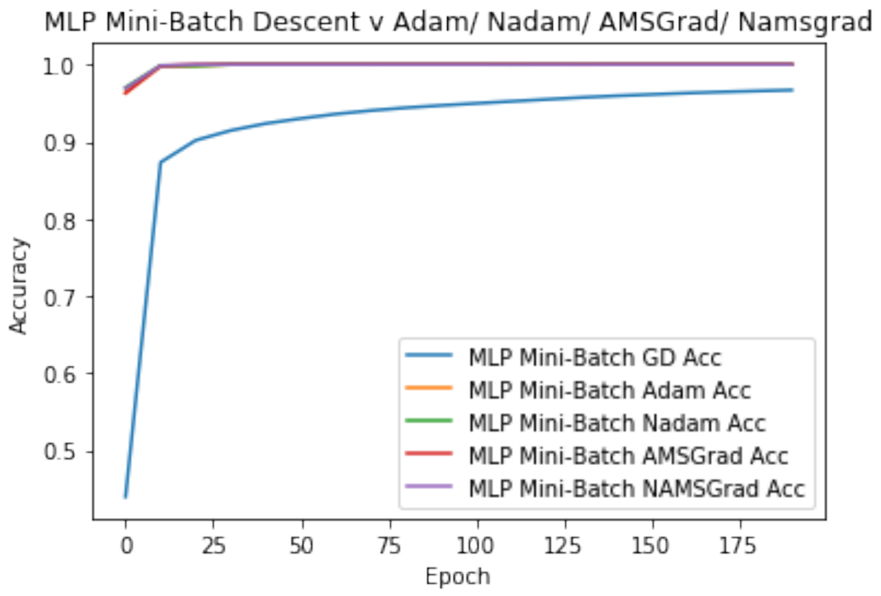


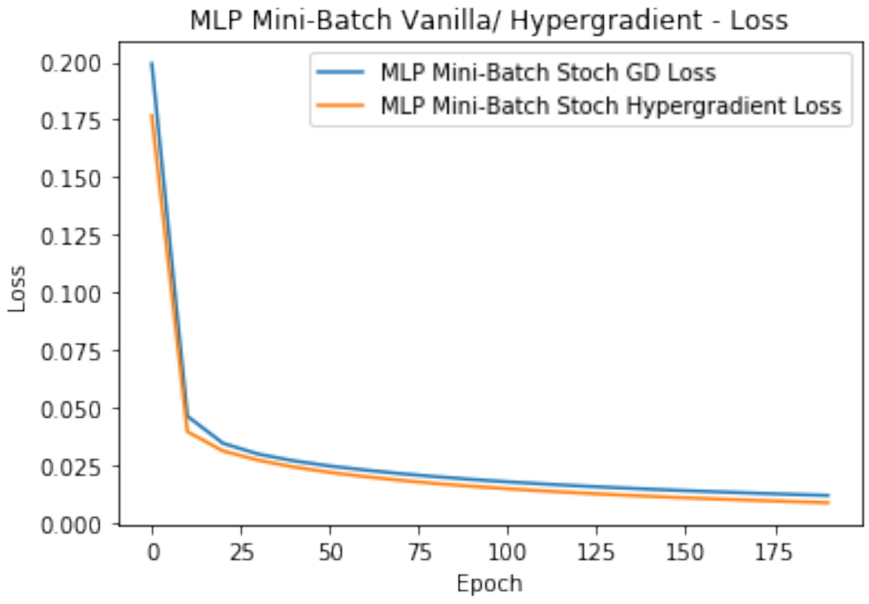
Figure 14: MLP - Min-Batch SGD Vanilla/ Adam/ Nadam/ AMSGrad/ NAMS-Grad

6.5.3 Mini-batch Gradient Descent - Hypergradient Descent

Figure 15 again shows hypergradient descent against plain vanilla gradient descent (both using mini-batches). The hypergradient has a small edge in this experiment, however I did only allow a low value for the learning rate of the hyper-gradient (the paper like much of this field does not appear to give many convergence guarantees, parts of the maths appear to demand a small β, and I didn't have time to experiment so kept things tight.

6.5.4 A closer examination of the best

Figure 16, shows an epoch by epoch evaluation of the 'best' optimisers. With mini-batch gradient descent on 50,000 training images, all achieved losses of zero and perfect accuracy on the training set, on the unseen test set they 'peaked out' at just over 98% accuracy. This was within 25 epochs. Indeed NAMSGrad took less than 20 epochs or only 1 minute of training on a quad core 3Ghz machine. The differences in behaviour between the algorithms were slight, although for this particular mini-batch gradient descent test Adam did seem to be slightly more volatile. My own silly attempt at Nesterov combined with new AMSGrad, at least in this context, performed at or close to the best. However, this is just an example illustration, not a conclusion.



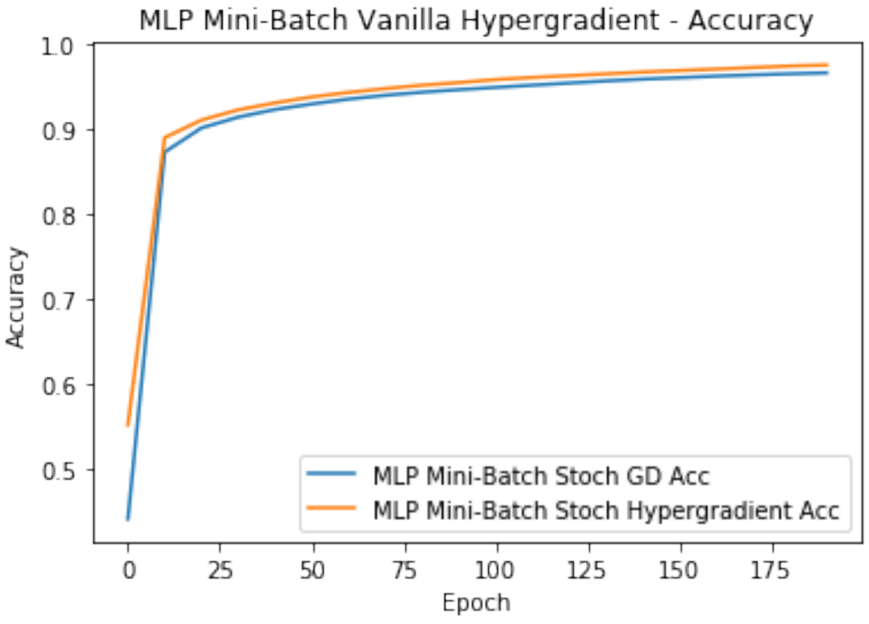
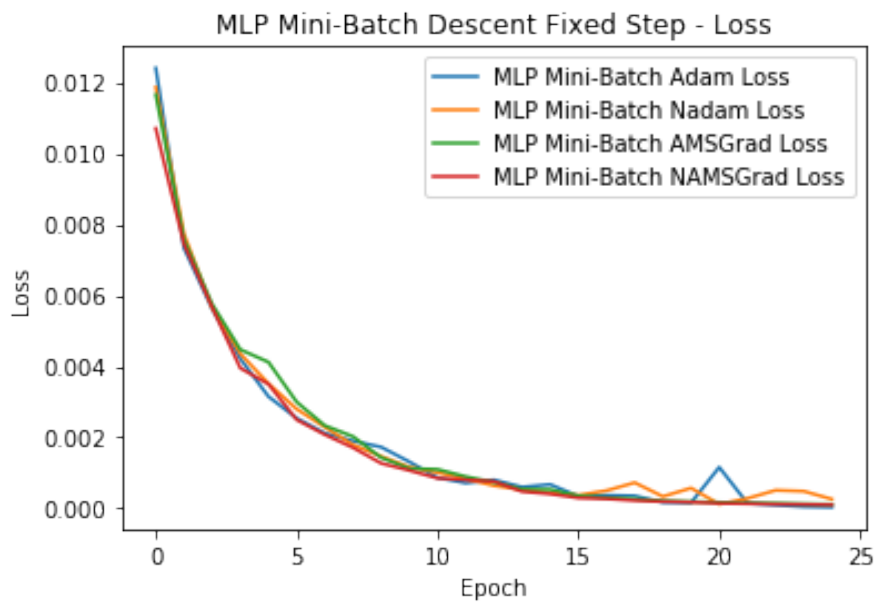


Figure 15: MLP - Mini-Batch Hypergradient Descent



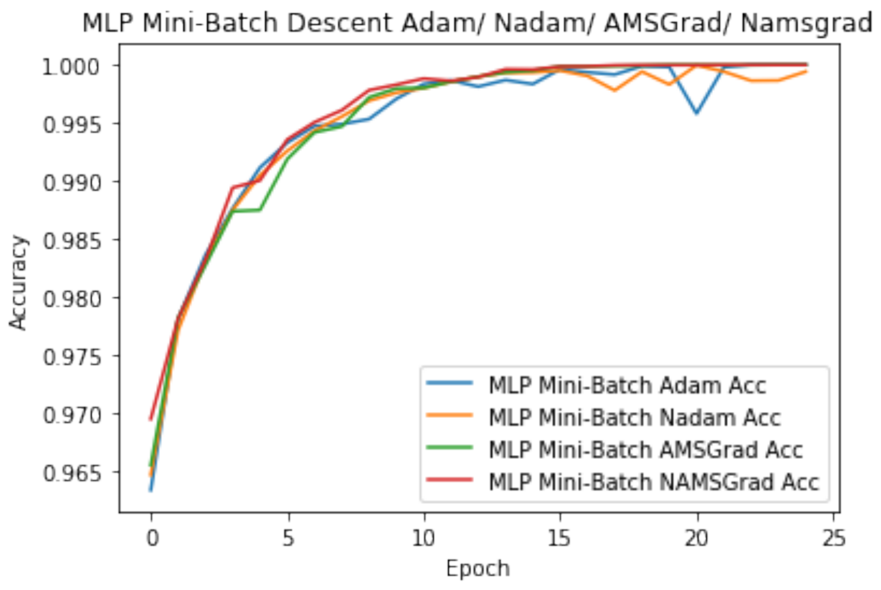


Figure 16: MLP - Adam/ Nadam/ AMSGrad/ NAMSGrad - incredible results in only a few epochs

7 Conclusion

Any conclusion is framed by the fact that everything in this document is associated with a single classification task. Some neural network architectures and a variety of optimisers were implemented. Including some very new ones, such as Hypergradient descent and AMSGrad, plus my somewhat obvious trick of adding Nesterov to AMSGrad. Within the limits of this task, and the sparsity of theoretical guarantees I demonstrated the following:

- Gradient Descent - Inefficient, all the data has to be examined each epoch. Slow to learn, scales badly
- Stochastic Gradient Descent - Can learn from a very small number of examples, learns in an online fashion, scalable, higher variance for fixed step.
- Mini-Batch Gradient Descent (appears to have the best of both worlds, quick updates, scalable, data efficient). Far better results by epoch.
- Architectures - Affine plus softmax lacks enough power. MLP is fast and more powerful than expected given the right optimiser. CNN very slow to train (lots of computation of kernels), but a powerful classifier with only a few epochs.
- How a variable learning rate can improve SGD stability
- The effect of L2 regularisation on the norm of weights and learning
- The differences (or in some cases little practical difference) between a variety of optimisers.
- The dramatic efficiency gains in this context of optimisers such as AMSGrad, Adam, Nadam and NAMSgrad.
- An implementation of hypergradient descent showing slight benefits, but my belief that the implementation can possibly be extended, and my conservatism using the hypergradient learning rate perhaps deserving further investigation.
- Approach - Basically I formulated various classifiers, an error function, and gradients with respect to the error function. This was first shown mathematically in this document and then implemented in code without libraries.

At the outset I intended to do further tests, particularly showing sample efficiency and training time and perhaps even moving away from pure optimisation to related results for generalisation in this context [6]. However, this is already too big, so I shall leave it there.

References

[1] Atilim Gunes Baydin et al. “Online Learning Rate Adaptation with Hypergradient Descent”. In: *arXiv preprint arXiv:1703.04782* (2017).

[2] Aleksandar Botev, Guy Lever, and David Barber. “Nesterov’s accelerated gradient and momentum as approximations to regularised update descent”. In: *Neural Networks (IJCNN), 2017 International Joint Conference on.* IEEE. 2017, pp. 1899–1903.

[3] Léon Bottou. “Stochastic gradient descent tricks”. In: *Neural networks: Tricks of the trade.* Springer, 2012, pp. 421–436.

[4] Léon Bottou. “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 0.

[5] Timothy Dozat. “Incorporating nesterov momentum into adam”. In: (2016).

[6] Moritz Hardt, Benjamin Recht, and Yoram Singer. “Train faster, generalize better: Stability of stochastic gradient descent”. In: *arXiv preprint arXiv:1509.01240* (2015).

[7] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[8] Yann LeCun. “The MNIST database of handwritten digits”. In: *http://yann. lecun. com/exdb/mnist/* (1998).

[9] Yurii Nesterov. “A method of solving a convex programming problem with convergence rate O (1/k2)”. In:

[10] Ning Qian. “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1 (1999), pp. 145–151.

[11] Alexander Rakhlin, Ohad Shamir, Karthik Sridharan, et al. “Making Gradient Descent Optimal for Strongly Convex Stochastic Optimization.” In: *ICML.* Citeseer. 2012.

[12] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. “On the convergence of adam and beyond”. In: *International Conference on Learning Representations.* 2018.

[13] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[14] Ashia C Wilson et al. “The marginal value of adaptive gradient methods in machine learning”. In: *Advances in Neural Information Processing Systems.* 2017, pp. 4151–4161.