How we imagined the working of the brain - 90s Edition.

Ajitesh Gupta, Jean Choi

Abstract. In this assignment we learned and implemented multilayer neural networks and learned about their working on the MNIST dataset.

For part 1, where we used one hidden layer, we achieved **92.31%** for using batch process. **400** hidden layer units were used throughout the experiments. $50,000$ training images, $10,000$ validation images and $10,000$ testing images were used.

For part 2, the accuracies improved progressively as **92.31%**, **94.15%**, **94.45%**, **97.7%** as we use mini-batch and weight normalization, tanh activation, regulated weight initialization, momentum respectively.

For part 3, we get accuracies of **97.72%**, **85.43%** for networks with **800** and **5** neurons respectively. It also gets an accuracy of **97.7%** for a double hidden layer network with **300** and **270** neurons each.

1 Introduction

1.1 Classification

Here we use a neural network with a single hidden layer to perform classification on the MNIST dataset. We have varied the size of the hidden layer between 300 and 800 neurons and report the results for the same. For this part we used sigmoid activation in the hidden layers and softmax activation in the output layer. Since nothing is specified and we have to compare with mini-batch performance, here we train using stochastic gradient descent.

The implementation is based on the following notations and equations:

Denote J as the number of units in hidden layer, C as the number of classes, and d as the dimension of the input, X.

For output layer, the output is given by

$$y_k = \frac{e^{sum_k}}{\sum_{k'} e^{sum_{k'}}} \tag{1}$$

and

$$E = -\sum_{l=1}^{C} t_l \log y_l \tag{2}$$

For hidden layer, the activation value of each unit is given by

$$z_j = \frac{1}{1 + e^{sum_j}} \tag{3}$$

Then

$$\delta_k = t_k - y_k \tag{4}$$

and

$$\delta_j = (1 - z_j) z_j \sum_{k=1}^{C} \delta_k w_{jk} \tag{5}$$

The weight update rules are:

$$w_{jk} = w_{jk} + \eta \delta_k z_j \tag{6}$$

$$w_{ij} = w_{ij} + \eta \delta_j x_i \tag{7}$$

1.2 "Tricks of the Trade"

Here we again conduct experiments using a standard 2 layer neural network architecture. But this time training is done in a mini-batch manner, in order to provide a speed boost. Also various optimizations are applied in the form of regulating the initial weights, using tanh as the activation function in the hidden layer, averaging the change in the weights over number of samples.

Since we use tanh as the activation function now, the corresponding δ_j is as follows :

$$\delta_j = 1.154 * (1 - \tanh^2(\frac{2}{3} z_j)) \sum_{k=1}^{C} \delta_k w_{jk} \tag{8}$$

1.3 Experiments with network topology

Here we saw the change in performance of the network by conducting 2 experiments with the structure of the network itself.

1.3.1 Doubling neurons

We double the number of neurons in the hidden layer i.e. double the number of parameters to be learned. We want to see whether increasing the number of parameters learned by the network increases the accuracy of the network and also what are its effects on the speed of the learning process itself.

1.3.2 Doubling hidden layers

We double the number of hidden layers while keeping similar number of parameters. We again want to see whether the increased complexity obtained by adding a second hidden layer makes the network learn to predict better and also again what is its effect on the learning speed. Using assumptions used in previous derivaitons and using i, j, k, l to denote input, hidden layer 1, hidden layer 2 and output layer and $H2$ to denote number of units in hidden layer 2 we get :

$$y_l = \frac{e^{sum_l}}{\sum_{l'} e^{sum_{l'}}} \tag{9}$$

$$z_k = 1.732 * \tanh\left(\frac{2}{3} sum_k\right) \tag{10}$$

$$z_j = 1.732 * \tanh\left(\frac{2}{3} sum_j\right) \tag{11}$$

then

$$\delta_l = t_l - y_l \tag{12}$$

$$\delta_k = 1.154 * (1 - \tanh^2(\frac{2}{3} z_k)) \sum_{l=1}^{C} \delta_l w_{kl} \tag{13}$$

$$\delta_j = 1.154 * (1 - \tanh^2(\frac{2}{3} z_j)) \sum_{k=1}^{H2} \delta_k w_{jk} \tag{14}$$

The weight update rules are:

$$w_{kl} = w_{kl} + \eta \delta_l z_k \tag{15}$$

$$w_{jk} = w_{jk} + \eta \delta_k x_j \tag{16}$$

$$w_{ij} = w_{ij} + \eta \delta_j x_i \tag{17}$$

2 Methods

2.1 Classification

The images were read in and normalized to a range of [0,1] by element-wise division by 255. Then they were mean subtracted in order to center the data.

As mentioned before, logistic activation was used for hidden layer while the output layer used softmax activation.

The parameters used in gradient descent such as number of epochs, m, initial learning rate, nu, and metaparameter, T, were given values based on experiments.

The accuracies were continuously monitored at each iteration and each iteration was timed in order to calculate the training time.

The following was the training procedure :

1. Normalize each sample by dividing by 255.
2. Center each image by subtracting mean.
3. Initialize weights by sampling randomly from a normal distribution with mean 0 and standard deviation of 1.
4. In each epoch for each sample do a forward pass and store the error.
5. At the end of the epoch, update the weights using the accumulated error.
6. Calculate the classification accuracy once the weights are updated.
7. Repeat till convergence.

Also to evaluate the correctness of the backprop we compare backprop updates with the ones obtained using the equation given in the question. For this we iterated through each weight in the network and did a forward pass with its original value say x, then with value x+ϵ and then with value x-ϵ. We used the normal weight to obtain backprop update and the results from the modified weights to calculate the numerical gradient. For each weight we keep summing the difference between the backprop update and the numerical gradient. Finally we average the sum over the number of weights to get the average difference.

2.2 "Tricks of the Trade"

In this part we incorporated various optimizations based on the reading of the paper by Yann LeCun.

- Mini Batch
  Since batch learning is not so accurate and stochastic learning takes time to iterate over the whole data, we use an in-between solution of mini-batches.

This helps in accurate and speedy learning. For the purpose of this assignment, we used mini-batches of size 125. We also averaged the weight update over the number samples in the batch.

- tanh Activation
  We used tanh activation in the hidden layer. This lead to a modified update equation as $\frac{d}{dx}\tanh(x)$ is $1 - \tanh^2(x)$.
- (Weight initialization)
  For each unit in the hidden layer and the output layer, we sample the initial weights from a normal distribution with mean of 0 and standard deviation of $\frac{1}{\sqrt{fan-in}}$ where fan-in stands for the number of inputs to that unit.
- Momentum
  Momentum term α used similar to its meaning in the physics world. Momentum involves using part of the previous update along with current update at any given iteration. It helps to avoid getting stuck in plateaus.

$$w(t+1) = w(t) + \alpha\Delta w(t-1) + \eta\Delta w(t) \tag{18}$$

So the final training procedure is as follows :

1. Normalize each sample by dividing by 255.
2. Center each image by subtracting mean.
3. Shuffle the examples.
4. Initialize weights by sampling randomly from a normal distribution with mean 0 and standard deviation of $\frac{1}{\sqrt{fan-in}}$.
5. For each mini-batch do a forward pass and calculate the error.
6. Normalize the error using the mini-batch size.
7. Update the weights using the calculated error and the also the previous error weighed by the momentum.
8. Calculate the classification accuracy after each epoch.
9. Repeat till convergence.

2.3 Experiments with Network Topology

- Doubling Neurons
  In this experiment we see the effect of doubling the number of neurons in the hidden layer of the network. In the previous networks we were using 400 neurons in the hidden layer so in this case we use 800 neurons. We

measure the time taken in each epoch along with the classification accuracy.

- Doubling the number of hidden layers
  Unlike normal 2 layer neural networks here we add another hidden layer. Again since we were using 400 neurons in the hidden layer leading to 318000 parameters, here we use 300 and 270 neurons in the two hidden layers leading to a very similar 319200 parameters. Again we measured both accuracy and time taken in each epoch.

  Owing to the added layer the new update equations are as follows :

3 Results and Discussions

3.1 Classification

For the full batch training we used $\eta = 0.1$ was used. The graph below shows the training accuracies for training, validation and testing sets as a function of number of epochs. The final accuracy achieved after 240 epochs was 92.31%. Also each epoch took around 1.5 seconds.

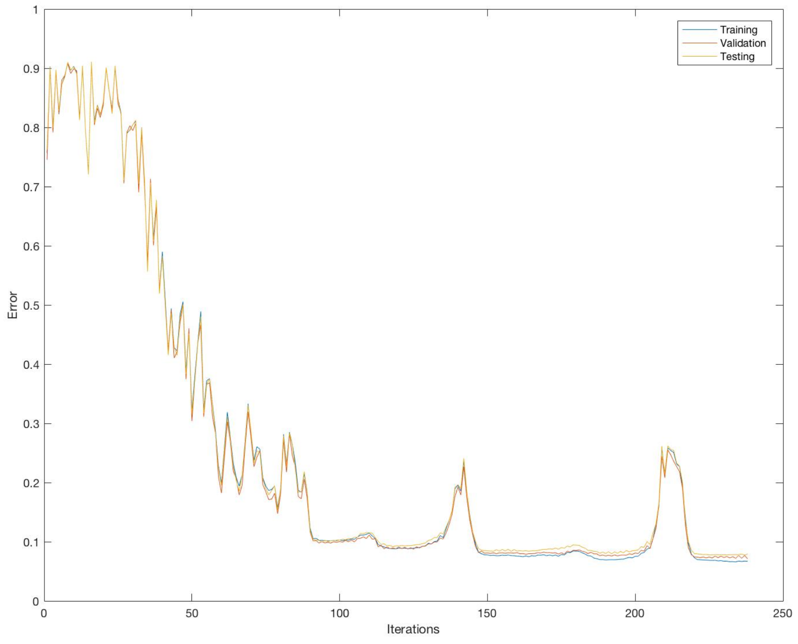

Figure 1: Standard full batch learning

As we can see the learning is very erratic owing to no normalization of the weight updates. Also the accuracy is only 5% better than the simple softmax regressor we used in the previous assignment.

Also regarding the correctness of the backprop update, we see that the average difference comes out to nearly zero with $\epsilon = 0.1$ and $\epsilon = 0.01$ and so on (ex. with $\epsilon = 0.01$ sum of differences comes out to 0.681 and averaging that over $3 * 10^6$ gives $2 * 10^{-7}$). Thus it is within the big-O of ϵ^2. Hence our backprop implementation is correct.

3.2 ”Tricks of the Trade”

(a) **Mini-batches and update normalization**

We used mini-batches of size 125 for convenience instead of 128. The error vs epoch graph for this method is as shown below.

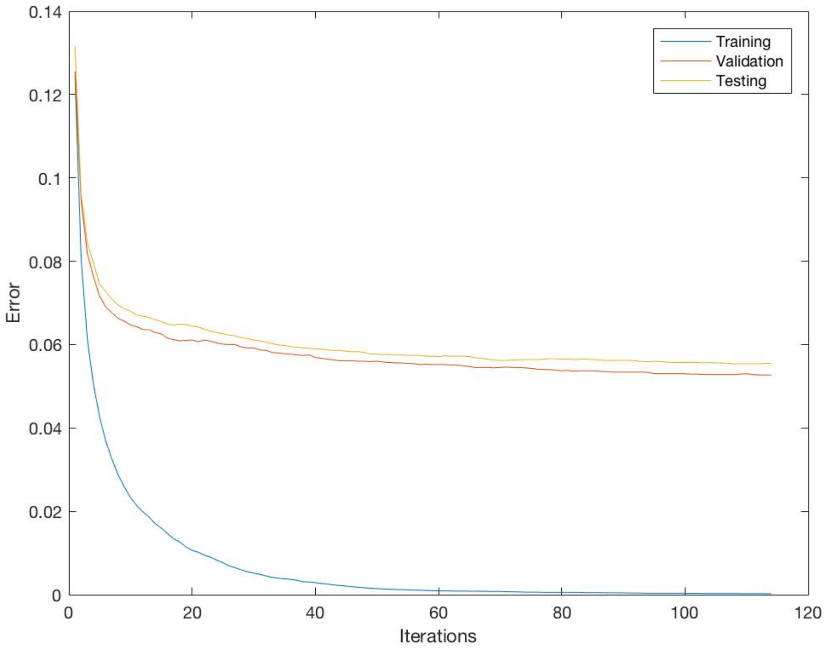

Figure 2: Mini batch learning

We can clearly see a marked improvement in the learning process. It is much more smoother than the previous batch method. We converge to 94.15% accuracy in just around 120 epochs. Also each epoch only takes 1.33 seconds. Thus mini-batches produce more accurate results than full batch while also being faster.

This is because the mini-batches retain qualities of both stochastic and batch gradient descent. The batch size being small makes updates faster and hence faster learning. Batches also means the updates are meaningful and not affected by random samples. Shuffling the samples also means that with high probability all successive samples belong to different classes.

(b) tanh **Activation**
Retaining the above modifications we further replaced the logistic activations in the hidden layer with $tanh$ activation function. Below we see the error vs epochs graph of the same.

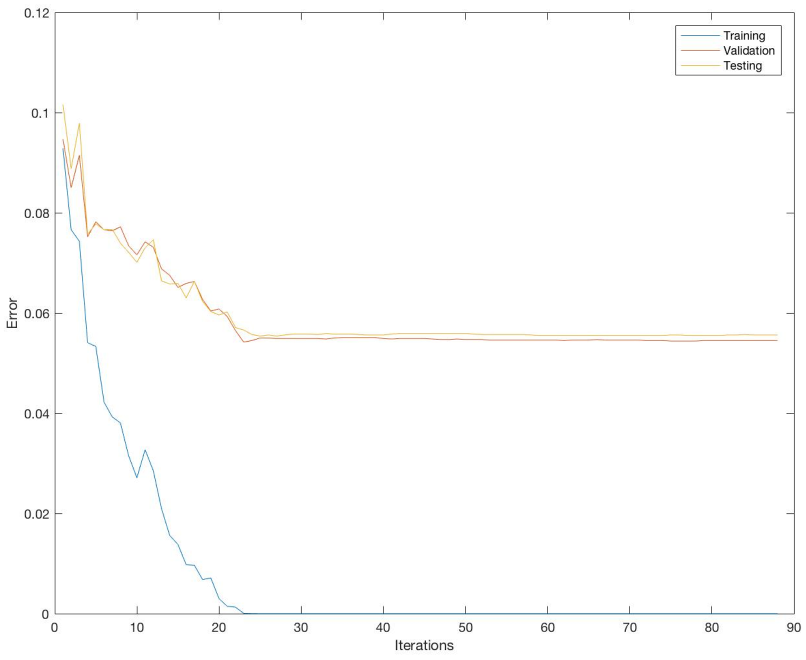

Figure 3: Learning using tanh activation

Again we can see the learning much stabler than the original full batch learning. It achieves a final accuracy of 94.45% and such the tanh function is better than the logistic activation function. Also although each epoch takes a little longer at 1.91 seconds seconds per epoch, it converges in around 80 epochs. As such it is still faster than the batch gradient descent.

(c) **Regulating initial weights**
Further we initialized the weights using a normal distribution of mean 0

and standard deviation equal to the inverse square root of the number of inputs in the corresponding neuron. Below is the corresponding error vs epoch graph.

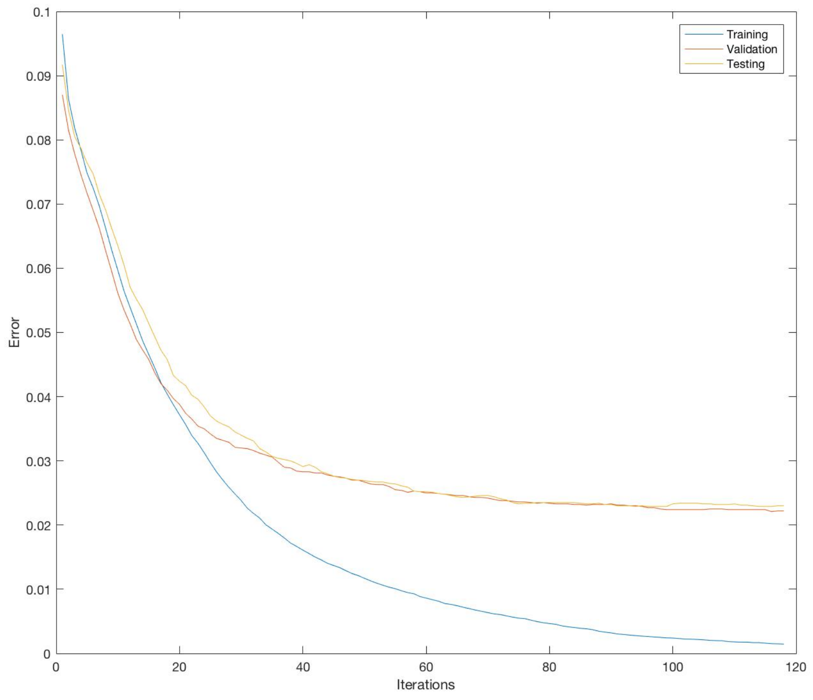

Figure 4: Effect of regulating initial weights

This method reaches a final accuracy of 97.7% in around 120 epochs and each epoch takes around 1.8 seconds. Thus the regulating the initial weights drastically improves the accuracy while also maintaining training speed.

(d) **Momentum**
For the experiments we used a momentum of 0.7. Below we see the graph of error vs epochs while using momentum.

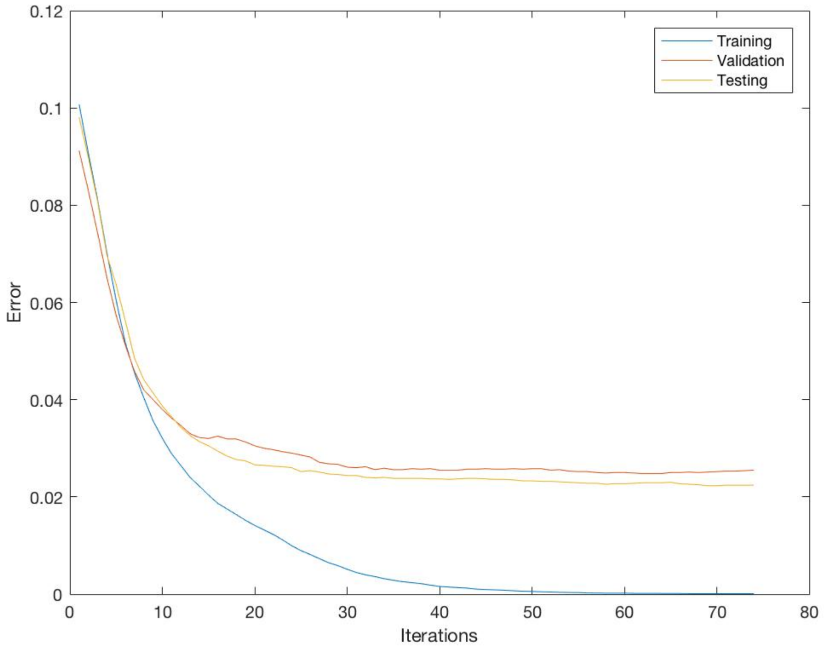

Figure 5: Learning with momentum

We see as expected the graph has a steeper slope meaning that the error goes down much faster. It reaches an accuracy of 97.77% in a much lesser 80 epochs. Thus again it generates better accuracy in faster time. The time taken per iteration is also around 2.07 seconds.

3.3 Experiments with network topology

(a) **Doubling number of neurons in hidden layer**
From the error vs epochs graph we can see that increasing the number of neurons did not have a profound effect on the training. The final training accuracy was again around 97.72% after around 100 iterations. Each epoch took around 3.23 seconds to process. As such it only leads to increased training time without much change in accuracy.

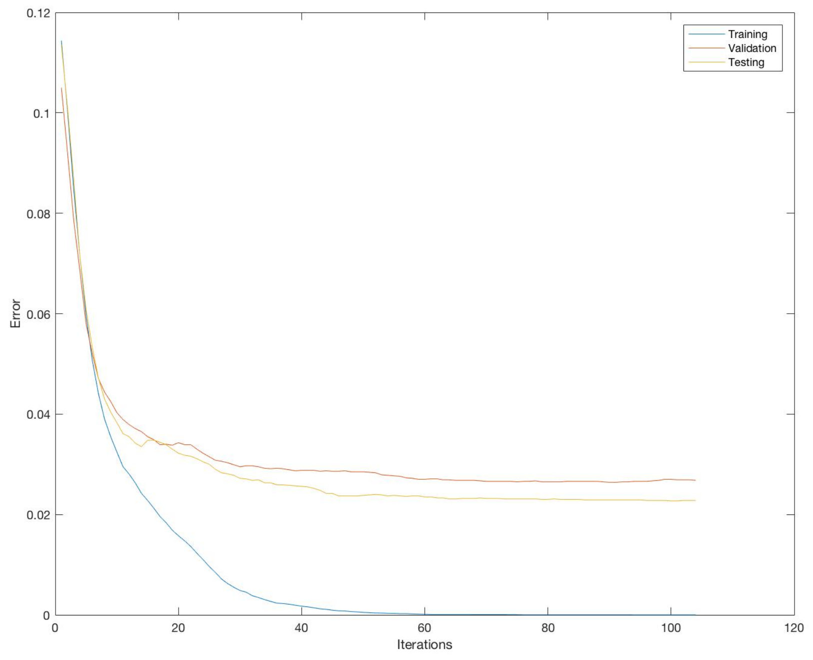

Figure 6: Learning with double the neurons

In another case we took 5 neurons to check effect of too low number of neurons. In this case each epoch was much faster at 0.45 seconds per epoch, but the accuracy took a hit. The accuracy achieved in this case was just 85.43%.

Thus in order to maintain a balance between accuracy and speed, we need to pick an optimum network size which is neither too big nor too small. Big networks lead to larger training times whereas smaller ones give lower accuracy.

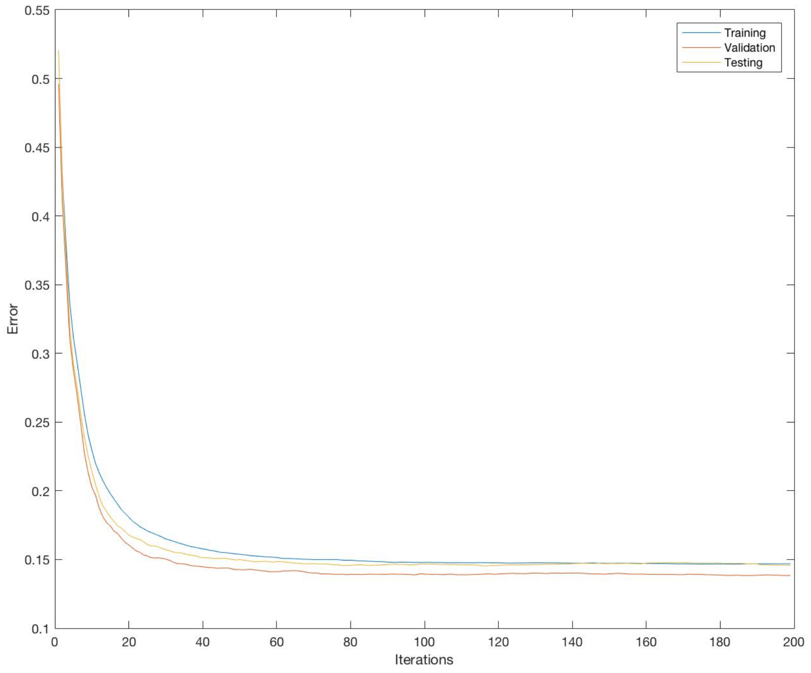

Figure 7: Learning with 5 neurons

(b) **Doubling hidden layers**
As we can see from the figure below, doubling the number of hidden layers again lead to similar accuracies of around 97.7% as achieved before. But it also takes around 150 epochs and each epoch takes around 3.17 seconds. Thus this is also unnecessary as although it might lead to a slightly better accuracy, it'll take much longer time to reach that. Only very complex classification problems can as such make use of the added complexity provided by an added hidden layer.

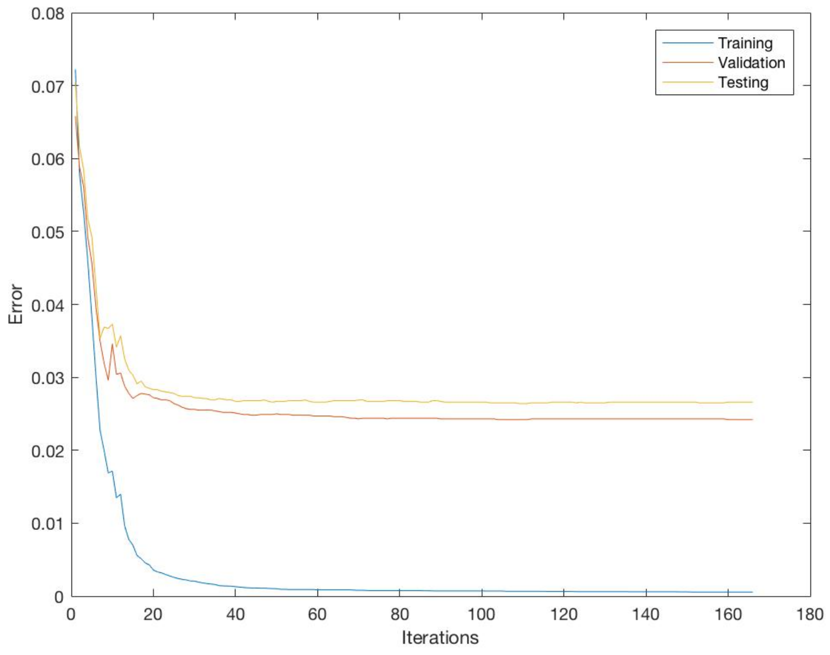

Figure 8: Learning with 2 hidden layers

4 Summary

(a) We see that neural networks work much better than the regression techniques we learnt in the last assignment.

(b) In particular we see that stochastic gradient descent leads to better learning than batch learning. A mid-way between the two can be in the form of mini-batches.

(c) Batch learning leads to low accuracy and the learning itself is also erratic.

(d) Mini-batches lead to a balance between fast learning and accurate learning.

(e) Normalizing the weight updates leads to faster learning and more accurate networks.

(f) A lot also depends on how we initialize our networks. Initializing weights to values, regulated by the number of their inputs leads to faster and more accurate learning.

(g) $1.732\tanh\left(\frac{2}{3}x\right)$ acts as a better activation function than regular logistic function as it is symmetrical around and centered at 0 whereas, the logistic function is always positive.

(h) Using momentum further speeds up the learning process and helps avoid plateaus.

(i) Needlessly increasing the number of neurons will just add to the overall training time whereas keeping the number too low will lead to lower learning.

(j) Increasing the number of layers might increase the accuracy in cases where the added level of complexity is required, but that is rare. So adding the extra layer again adds to training time while the accuracy remains nearly same as the network with single hidden layer.

5 Contributions

I, Jean Choi, did 3.
I, Ajitesh Gupta, did 4 and 5.

Appendix

```
% Change the filenames if you've saved the files under different
    names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
images_train = loadMNISTImages('train-images-idx3-ubyte')';
labels_train = loadMNISTLabels('train-labels-idx1-ubyte');
y_train = labels_train;
labels_train = onehot(labels_train,10);

for it=1:60000
    m = mean(images_train(it,:));
    images_train(it,:) = images_train(it,:)-m;
end

images_test = loadMNISTImages('t10k-images-idx3-ubyte')';
labels_test = loadMNISTLabels('t10k-labels-idx1-ubyte');
y_test = labels_test;
labels_test = onehot(labels_test,10);


for it=1:10000
    m = mean(images_test(it,:));
    images_test(it,:) = images_test(it,:)-m;
```

```
end

ntrain = 50000;
nval = 10000;
ntest = 10000;

train_x = [ones(ntrain,1),images_train(1:ntrain,:)];
train_y = labels_train(1:ntrain,:)';

val_x = [ones(nval,1),images_train(ntrain+1:ntrain+nval,:)];
val_y = labels_train(ntrain+1:ntrain+nval,:)';

test_x = [ones(ntest,1),images_test(1:ntest,:)];
test_y = labels_test(1:ntest,:)';

% Setting hyperparameters
n_iters = 200;
nh = 400;
ni = 784;
no = 10;
eta0 = 0.3;
eta = eta0;
T = 10000;
mom = 0.7;
trainsize = 125;

% Initialization considering fan-in
w1 = normrnd(0,1/sqrt(ni),[ni+1,nh]);
w1(end,:) = normrnd(0,1,[1,nh]);
w2 = normrnd(0,1/sqrt(nh),[nh+1,no]);
w2(end,:) = normrnd(0,1,[1,no]);

% Initialization without fan-in
% w1 = normrnd(0,1,[ni+1,nh]);
% w1(end,:) = normrnd(0,1,[1,nh]);
% w2 = normrnd(0,1,[nh+1,no]);
% w2(end,:) = normrnd(0,1,[1,no]);

% Shuffling data
r = randi([1 ntrain],1,ntrain);
train_x(1:ntrain,:) = train_x(r,:);
train_y(:,1:ntrain) = train_y(:,r);

tr_ac = zeros([1,n_iters]);
val_ac = zeros([1,n_iters]);
te_ac = zeros([1,n_iters]);
time = 0.0;
for i=1:n_iters
    tic;
    % Running mini-batch
    for k=1:ntrain/trainsize
        updatek = zeros([nh+1,no]);
        updatej = zeros([ni+1,nh]);
        start = (k-1)*trainsize;
        %forward propagate
        netj = train_x(start+1:start+trainsize,:)*w1;
        fnetj = [logistic(netj) ones([trainsize,1])];
        %fnetj = [1.732.*tanh(2/3.*netj) ones([trainsize,1])];
        netk = fnetj*w2;
        fnetk = softmax(netk);

        %error calc
        dk = (train_y(:,start+1:start+trainsize)-fnetk');
        u1 = updatek;
        updatek = updatek + (fnetj'*dk');
        dj = (dk'*w2').*fnetj.*(1 - fnetj);
        %dj = (dk'*w2').*1.154.*(1 - tanh(2/3.*fnetj).*tanh(2/3.*
    fnetj));
        u2 = updatej;
        updatej = updatej + train_x(start+1:start+trainsize,:)'*dj
    (:,1:end-1);

        w2 = w2 + mom.*u1 + eta.*updatek./trainsize;
        w1 = w1 + mom.*u2 + eta.*updatej./trainsize;
        eta = eta0/(1+i/T);
    end
    time = time+toc;
    % Checking accuracies
    netj = train_x(1:ntrain,:)*w1;
    fnetj = [logistic(netj) ones([ntrain,1])];
    %fnetj = [1.732.*tanh(2/3.*netj) ones([ntrain,1])];
    netk = fnetj*w2;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(train_y(:,1:ntrain),[],1);
    %for x=1:nval
    %    fprintf('%d %d %d\n',Idx(x),Idx2(x),y_train(ntrain+x));
    %end
    count0 = nnz(Idx==Idx2');

    netj = val_x(1:nval,:)*w1;
    fnetj = [logistic(netj) ones([nval,1])];
    %fnetj = [1.732.*tanh(2/3.*netj) ones([nval,1])];
    netk = fnetj*w2;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(val_y(:,1:nval),[],1);
    %for x=1:nval
    %    fprintf('%d %d %d\n',Idx(x),Idx2(x),y_train(ntrain+x));
    %end
    count1 = nnz(Idx==Idx2');

    netj = test_x(1:ntest,:)*w1;
    fnetj = [logistic(netj) ones([ntest,1])];
    %fnetj = [1.732.*tanh(2/3.*netj) ones([ntest,1])];
    netk = fnetj*w2;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(test_y(:,1:ntest),[],1);
    %fprintf('%d %d %d\'n,Idx,Idx2,y_test);
    count2 = nnz(Idx==Idx2');
    tr_ac(i) = count0/ntrain;
    val_ac(i) = count1/nval;
    te_ac(i) = count2/ntest;
    fprintf('Iteration : %d Train Accuracy : %0.4f Val Accuracy :
    %0.4f Test Accuracy : %0.4f Time : %0.2f\n',i,count0/ntrain,
    count1/nval,count2/ntest,time/i);
end

function [g] = logistic(z)
par_res=1+exp(-z);
g=1./par_res;
end

function [g] = tanhh(z)
g = 2/3.*tanh(z);
end

function [softmaxA] = softmax(A)
dim = 2;
s = ones(1, ndims(A));
s(dim) = size(A, dim);
maxA = max(A, [], dim);
expA = exp(A-repmat(maxA, s));
softmaxA = expA ./ repmat(sum(expA,dim), s);
end

function [oh] = onehot(v,c)
oh = zeros(length(v), c);
for i=1:length(v)
    oh(i,v(i)+1) = 1;
end
end
```

Listing 1: MATLAB code to run a 2 layer neural network

```
% Change the filenames if you've saved the files under different
    names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
```

```
images_train = loadMNISTImages('train-images-idx3-ubyte')';
labels_train = loadMNISTLabels('train-labels-idx1-ubyte');
y_train = labels_train;
labels_train = onehot(labels_train,10);

for it=1:60000
    m = mean(images_train(it,:));
    images_train(it,:) = images_train(it,:)-m;
end

images_test = loadMNISTImages('t10k-images-idx3-ubyte')';
labels_test = loadMNISTLabels('t10k-labels-idx1-ubyte');
y_test = labels_test;
labels_test = onehot(labels_test,10);


for it=1:10000
    m = mean(images_test(it,:));
    images_test(it,:) = images_test(it,:)-m;
end

ntrain = 50000;
nval = 10000;
ntest = 10000;

train_x = [ones(ntrain,1),images_train(1:ntrain,:)];
train_y = labels_train(1:ntrain,:)';

val_x = [ones(nval,1),images_train(ntrain+1:ntrain+nval,:)];
val_y = labels_train(ntrain+1:ntrain+nval,:)';

test_x = [ones(ntest,1),images_test(1:ntest,:)];
test_y = labels_test(1:ntest,:)';

% Setting hyperparameters
n_iters = 20000;
nh1 = 300;
nh2 = 270;
ni = 784;
no = 10;
eta0 = 0.3;
eta = eta0;
T = 1000000000;
trainsize = 50;
mom = 0.7;

% Initialization considering fan-in
w1 = normrnd(0,1/sqrt(ni),[ni+1,nh1]);
w1(end,:) = normrnd(0,1,[1,nh1]);
w2 = normrnd(0,1/sqrt(nh1),[nh1+1,nh2]);
```

```
w2(end,:) = normrnd(0,1,[1,nh2]);
w3 = normrnd(0,1/sqrt(nh2),[nh2+1,no]);
w3(end,:) = normrnd(0,1,[1,no]);

tr_ac = zeros([1,n_iters]);
val_ac = zeros([1,n_iters]);
te_ac = zeros([1,n_iters]);

for i=1:n_iters
    for k=1:ntrain/trainsize
        updatek = zeros([nh2+1,no]);
        updatej = zeros([nh1+1,nh2]);
        updatei = zeros([ni+1,nh1]);
        start = (k-1)*trainsize;

        %forward propagate
        neti = train_x(start+1:start+trainsize,:)*w1;
        %fneti = [sigmoid(neti) ones([trainsize,1])];
        fneti = [tanh(1.732.*2/3.*neti) ones([trainsize,1])];
        netj = fneti*w2;
        %fnetj = [sigmoid(netj) ones([trainsize,1])];
        fnetj = [1.732.*tanh(2/3.*netj) ones([trainsize,1])];
        netk = fnetj*w3;
        fnetk = softmax(netk);

        %error calc
        dk = (train_y(:,start+1:start+trainsize)-fnetk');
        u1 = mom.*updatek;
        updatek = updatek + (fnetj'*dk');
        %dj = (dk'*w3').*fnetj.*(1-fnetj);
        dj = (dk'*w3').*(1-tanh(fnetj).*tanh(fnetj));
        u2 = mom.*updatej;
        updatej = updatej + fneti'*dj(:,1:end-1);
        %di = (dj(:,1:end-1)*w2').*fneti.*(1-fneti);
        di = (dj(:,1:end-1)*w2').*1.154.*(1 - tanh(2/3.*fneti).*tanh
    (2/3.*fneti));
        u3 = mom.*updatei;
        updatei = updatei + train_x(start+1:start+trainsize,:)'*di
    (:,1:end-1);
        w3 = w3 + u1+eta.*updatek./trainsize;
        w2 = w2 + u2+eta.*updatej./trainsize;
        w1 = w1 + u3+eta.*updatei./trainsize;
        eta = eta0/(1+i/T);

    end
    % Calculate accuracies
    neti = train_x(1:ntrain,:)*w1;
    %fneti = [sigmoid(neti) ones([ntrain,1])];
    fneti = [1.732.*tanh(2/3.*neti) ones([ntrain,1])];
    netj = fneti*w2;
    %fnetj = [sigmoid(netj) ones([ntrain,1])];
    fnetj = [1.732.*tanh(2/3.*netj) ones([ntrain,1])];
    netk = fnetj*w3;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(train_y(:,1:ntrain),[],1);
    count0 = nnz(Idx==Idx2');

    neti = val_x(1:nval,:)*w1;
    %fneti = [sigmoid(neti) ones([nval,1])];
    fneti = [1.732.*tanh(2/3.*neti) ones([nval,1])];
    netj = fneti*w2;
    %fnetj = [sigmoid(netj) ones([nval,1])];
    fnetj = [1.732.*tanh(2/3.*netj) ones([nval,1])];
    netk = fnetj*w3;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(val_y(:,1:nval),[],1);
    count1 = nnz(Idx==Idx2');

    neti = test_x(1:ntest,:)*w1;
    %fneti = [sigmoid(neti) ones([ntest,1])];
    fneti = [1.732.*tanh(2/3.*neti) ones([ntest,1])];
    netj = fneti*w2;
    %fnetj = [sigmoid(netj) ones([ntest,1])];
    fnetj = [1.732.*tanh(2/3.*netj) ones([ntest,1])];
    netk = fnetj*w3;
    fnetk = softmax(netk);
    [~, Idx] = max(fnetk,[],2);
    [~, Idx2] = max(test_y(:,1:ntest),[],1);
    count2 = nnz(Idx==Idx2');
    tr_ac(i) = count0/ntrain;
    val_ac(i) = count1/nval;
    te_ac(i) = count2/ntest;
    fprintf('Iteration : %d     Train Accuracy : %0.4f Val
    Accuracy : %0.4f     Test Accuracy : %0.4f\n',i,count0/ntrain,
    count1/nval,count2/ntest);
end

function [g] = sigmoid(z)
par_res=1+exp(-z);
g=1./par_res;
end

function [softmaxA] = softmax(A)
dim = 2;
s = ones(1, ndims(A));
s(dim) = size(A, dim);
maxA = max(A, [], dim);
expA = exp(A-repmat(maxA, s));
```

```
softmaxA = expA ./ repmat(sum(expA,dim), s);
end

function [oh] = onehot(v,c)
oh = zeros(length(v), c);
for i=1:length(v)
    oh(i,v(i)+1) = 1;
end
end
```

Listing 2: MATLAB code to run a 3 layer neural network

```
function images = loadMNISTImages(filename)
%loadMNISTImages returns a 28x28x[number of MNIST images] matrix
    containing
%the raw MNIST images

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2051, ['Bad magic number in ', filename, '']);

numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

images = fread(fp, inf, 'unsigned char');
images = reshape(images, numCols, numRows, numImages);
images = permute(images,[2 1 3]);

fclose(fp);

% Reshape to #pixels x #examples
images = reshape(images, size(images, 1) * size(images, 2), size(
    images, 3));
% Convert to double and rescale to [0,1]
images = double(images) / 255;

end
```

Listing 3: MATLAB code to load images from data

```
function labels = loadMNISTLabels(filename)
%loadMNISTLabels returns a [number of MNIST images]x1 matrix
    containing
%the labels for the MNIST images

fp = fopen(filename, 'rb');
assert(fp ~= -1, ['Could not open ', filename, '']);
```

```

magic = fread(fp, 1, 'int32', 0, 'ieee-be');
assert(magic == 2049, ['Bad magic number in ', filename, '']);

numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');

labels = fread(fp, inf, 'unsigned char');

assert(size(labels,1) == numLabels, 'Mismatch in label count');

fclose(fp);

end
```

Listing 4: MATLAB code to load labels from data

```
% Change the filenames if you've saved the files under different
    names
% On some platforms, the files might be saved as
% train-images.idx3-ubyte / train-labels.idx1-ubyte
images_train = loadMNISTImages('train-images-idx3-ubyte')';
labels_train = loadMNISTLabels('train-labels-idx1-ubyte');
y_train = labels_train;
labels_train = onehot(labels_train,10);

for it=1:60000
    m = mean(images_train(it,:));
     images_train(it,:) = images_train(it,:)-m;
end

images_test = loadMNISTImages('t10k-images-idx3-ubyte')';
labels_test = loadMNISTLabels('t10k-labels-idx1-ubyte');
y_test = labels_test;
labels_test = onehot(labels_test,10);


for it=1:10000
    m = mean(images_test(it,:));
     images_test(it,:) = images_test(it,:)-m;
end

ntrain = 50000;
nval = 10000;
ntest = 10000;

train_x = [ones(ntrain,1),images_train(1:ntrain,:)];
train_y = labels_train(1:ntrain,:)';

val_x = [ones(nval,1),images_train(ntrain+1:ntrain+nval,:)];
val_y = labels_train(ntrain+1:ntrain+nval,:)';
```

```

test_x = [ones(ntest,1),images_test(1:ntest,:)];
test_y = labels_test(1:ntest,:)';

load weights.mat;
eta = 0.001;
dnum = 0.0;
dcalc = 0.0;
count = 0;
epsilon = 0.01;
nlayers = length(w);
n = 100000;
delta = 0.0;
s = 1;
for i=1:nlayers
    for j=1:size(w{i},1)
        for k=1:size(w{i},2)
            %forward propagate
            wp = w;
            wn = w;
            wp{i}(j,k) = wp{i}(j,k)+epsilon;
            wn{i}(j,k) = wn{i}(j,k)-epsilon;

            netj = train_x(s,:)*w{1};
            fnetj = [sigmoid(netj) 1];
            netk = fnetj*w{2};
            y = softmax(netk);

            netj = train_x(s,:)*wp{1};
            fnetj = [sigmoid(netj) 1];
            netk = fnetj*wp{2};
            yp = softmax(netk);

            netj = train_x(s,:)*wn{1};
            fnetj = [sigmoid(netj) 1];
            netk = fnetj*wn{2};
            yn = softmax(netk);

            dnum = calcNumGradient(yp,yn,train_y(:,s),epsilon);

            %error calc
            update = w;
            dk = (train_y(:,s)-y');
            update{2} =(fnetj'*dk');
            dj = (dk'*w{2}').*fnetj.*(1-fnetj);
            update{1} = train_x(s,:)'*dj(:,1:end-1);
            dcalc = -update{i}(j,k);
            delta = delta + abs(dcalc-dnum);
            count = count+1;
        end
    end
end

fprintf('Avg Difference in Gradient : %0.4f     Epsilon square :
    %0.4f\n',delta/count,epsilon^2);

function [g] = sigmoid(z)
par_res=1+exp(-z);
g=1./par_res;
end

function [softmaxA] = softmax(A)
dim = 2;
s = ones(1, ndims(A));
s(dim) = size(A, dim);
maxA = max(A, [], dim);
expA = exp(A-repmat(maxA, s));
softmaxA = expA ./ repmat(sum(expA,dim), s);
end

function [oh] = onehot(v,c)
oh = zeros(length(v), c);
for i=1:length(v)
    oh(i,v(i)+1) = 1;
end
end

function [numE] = calcNumGradient(yp,yn,t,ep)
yp =log(yp);
yn = log(yn);
Ep = -yp*t;
En = -yn*t;
numE = (Ep-En)/(2*ep);
end
```

Listing 5: MATLAB code to verify the correctness of the backprop implementation