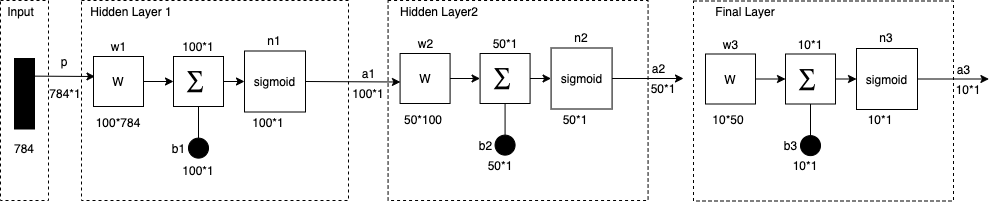
Fashion-MNIST Project

# Introduction

In this assignment, we further explore the backpropagation algorithm and techniques to improve the performance of our network such as mini-batching, stochastic descent, SoftMax, cross entropy, and momentum for faster training. Specifically, we work with a network similar to the one we designed for the backpropagation assignment, which was used to solve the MNIST digit classification problem.

Our network consists of 2 hidden layers, in which each layer has 100 and 50 neurons. The input layer has 784 input neurons, since each image has 784 pixels. There are 10 labels in the target set to classify, then the output layer will have 10 neurons to encode all 10 labels/classes. The diagram of our ANN is as follow:



*Figure 1 Diagram of our ANN*

# Methods

## Preprocessing data

We read the file ‘train.csv’ to a MATLAB table, then convert it to array and normalize the data, which makes all inputs in range of [0, 1] to reduce the cost of calculation. Then we make an array of dimension 60000\*784 to store all pixels of image from the training set and make an array of dimension 10\*60000 to store labels of all images from the training set. After doing a loop, we have the input and target matrices for training our network.

## FashionPreprocessing.m

function FashionPreprocessing() T = readtable('train.csv');

fashion = table2array(T);

f\_normalized= normalize(fashion,'range'); [r,c]=size(fashion);

inputs = zeros(r,c-2); targets = zeros(10,r);

% Write data to input and target matrices for i = 1:r

row = f\_normalized(i,3:c); target = fashion(i,2); inputs(i,:)=row;

targets(:,i)= fashion\_taget(target); end

% Write data to mat file save('fashion.mat','inputs','targets'); end

## Forward pass

The input will be forward through the network and return output vector of 10\*1 dimension. Since we are using sigmoid function for both hidden layers, we just call a for loop to accumulate output of each hidden layer to final layer. The parameters for this function is inputs data, number of hidden layers, weights\_array, which is a cell array containing weight matrices of all hidden layer, biases array, which is also a cell array of all biases of hidden layers.

## forward\_Multi.m

function [final\_hiddens,final\_output] = forward\_Multi(inputs,hidden\_layers,weights\_array,biases\_array)

% Initialize output of all layers final\_hiddens{1,hidden\_layers}=[]; net\_hidden=inputs;

% Calculate output of all hidden layers for i = 1 : hidden\_layers

value\_hidden = weights\_array{i,1}\* net\_hidden+biases\_array{i,1};

% Output of all hidden layers after applying sigmoid tmp\_hidden =sigmoid(value\_hidden); final\_hiddens{1,i}=tmp\_hidden; net\_hidden=tmp\_hidden;

end

% Output of the final layer

value\_final = weights\_array{1+hidden\_layers,1} \* net\_hidden + biases\_array{1+hidden\_layers,1};

final\_output = sigmoid(value\_final); end

## Loss function

We’re using quadratic loss function for our network. The cost is calculated as

cost =1/n \*(t -a)^2

Where n is number of elements in output vector, t is target vector and a is output vector.

## quaradic\_loss.m

function [cost] = quaradic\_loss(target,output) [r,c]=size(target);

cost=0; for i=1:r

t=target(i,1); p=output(i,1); cost=cost+0.5\*(t-p)^2; end

end

## Sensitive for backpropagation

We have derivative for cost function and derivative of activation function. Then we can calculate sensitive parameter for each layer

* + Derivative quadratic loss: (t – a)
  + Derivative sigmoid: sigmoid\_Prime(z) = sigmoid(z)\*(1 – sigmoid(z))
  + Sensitive:
    - Final layer:

S[3] = -2\*(t – a) \* sigmoid\_Prime(final output);

* + - Second hidden layer:

S[2] = sigmoid\_Prime(final\_hidden[2]) \* (weights[3]'\*S[3] )

* + - First hidden layer:

S[1] = sigmoid\_Prime(final\_hidden[1]) \* (weights[1]'\*S[2] )

## Backpropagation

For the first attempt of training our network, we’re using stochastic gradient descent (SGD), which means all weight and bias matrices are update for each example form the training set. For each epoch, we iterate through all examples. And in each iteration, we call backward pass to get the output of the sample with current network parameter (weights and biases).

Then we use those output to calculate the sensitives to update weight and bias matrices at the end of each iteration. This function returns the final weight and biases matrices and average error from loss function of the network.

## Backprop.m

function [weights,biases,arrayCost] =

backProp(inputs,targets,input\_neurons,hidden\_neurons,hidden\_neur ons2,output\_neurons,epoch, learning\_rate)

% Number of sample data k = size(inputs,1);

% Network parameter neurons=[input\_neurons,hidden\_neurons,hidden\_neurons2,output\_neu rons];

% Initialize weights and biases randomly

[weights,biases]=initWeights(2,neurons); arrayCost = zeros(epoch,1);

% Iteration epoch number for j = 1:epoch

j

% Iteration over all sample data for i = 1:k

% Feed forward [final\_hiddens,final\_output] = forward(inputs(i,:)',weights,biases);

% Calculate error

diff= (targets(:,i) - final\_output);

% Quadratic error

cost= quaradic\_loss(targets(:,i),final\_output); avgCost = avgCost + cost;

% Back-propagation start

% Calculate sensitive value for each layer s=sensitive(diff,final\_output,final\_hiddens{1,2},final\_hiddens{1

,1},weights);

% Update weight matrices

weights{1,1} = weights{1,1} - learning\_rate .\* s{1,1}\* inputs(i,:);

weights{2,1} = weights{2,1} - learning\_rate .\* s{1,2}\* final\_hiddens{1,1}';

weights{3,1} = weights{3,1} - learning\_rate .\* s{1,3}\* final\_hiddens{1,2}';

% Update bias

biases{1,1} = biases{1,1} -learning\_rate\*s{1,1}; biases{2,1} = biases{2,1} -learning\_rate\*s{1,2}; biases{3,1}= biases{3,1} -learning\_rate\*s{1,3};

% Back-propagation end end

avgCost = avgCost/k arrayCost(j)=avgCost; end

end

## Calculate Accuracy:

The likelihood of the output vector and target vector is determined by comparing the index of the largest element in each vector. If the 2 indexes are equal, then it’s considered a successful recognition.

From the weights and biases of our training, we can calculate the accuracy of our network on the training set. This function returns the percentage of success recognitions for testing a list of examples.

## Testset.m

function [percentage]=testSet(inputs, targets, weights, biases)

% inputs:set of sample to test

% targets: label of data\_set, index of largest value of an output set

success=0; [r,c]=size(inputs);

% Load weights and bias

% Loop through all samples for i = 1:r

t=test\_Multi(inputs(i,:)',2,weights, biases); if(getLargestIndex(t)==getLargestIndex(targets(:,i)))

success=success+1;

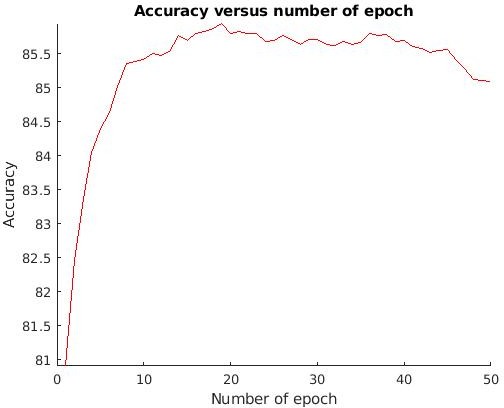
end

end

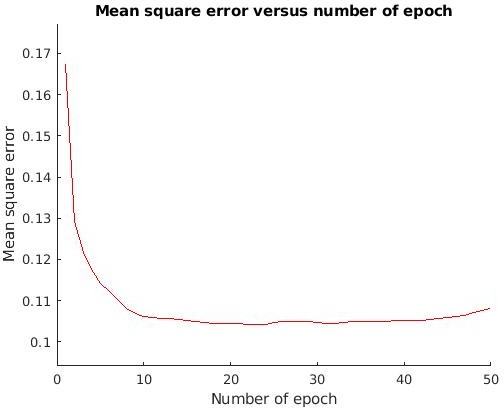
percentage=(success/r)\*100; end

# Results

We trained the network for 50 epochs. The accuracy and average cost are displays in the following diagrams



*Figure 2: Graph of accuracy versus number of epochs*



*Figure 3: Graph of mean square error versus number of epochs*

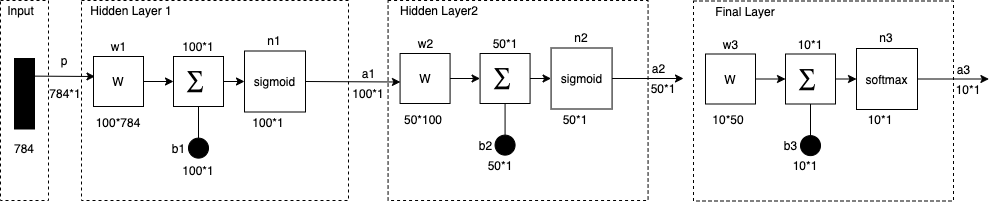
From the graphs, we see that our network seems to stop increasing accuracy and decreasing loss after epoch 35. That means it cannot convert to a minimum. Then we will try to improve the accuracy as well as reduce error in next section.

# Improve our network

We’re trying to improve our network with several methods such as cross entropy and SoftMax, mini-batch, and momentum.

## Using cross entropy loss function and SoftMax function for final layer

We’ll use SoftMax as transfer function for final layer. The diagram of our network now is as follow:



*Figure 4: Diagram of our ANN with SoftMax function at final layer*

## Cross entropy loss function:

The cross-entropy loss function is as follow:

= ∑& % ∗ log(%) + (1 − %) ∗ log(1 − %)

%'(

In which n is number of elements of the vector target or output (=10 in our network), t is target vector and y is output vector. Its code in MATLAB is:

## crossentropy\_loss.m

function [cost] = crossentropy\_loss(target,output) [r,c]=size(target);

cost=0; for i=1:r

t=target(i,1); p=output(i,1);

cost=cost+t\*log2(p) + (1-t)\*log2(1-p);

end

end

cost=cost\*-1;

## SoftMax transfer function

SoftMax function takes an N-dimensional vector of real numbers and transforms it into a vector of real number in range (0,1) which helps to highlight the index of the largest element for classification. Its formula is:

.

= 456

% 8

∑

79:

457

Its implementation is as follow

## softmax.m

function [output] = softmax(input)

mx=max(input); input=input-mx;

output=exp(input)/sum(exp(input)); end

## Derivative of cross entropy loss function with Softmax

The derivative of cross entropy loss function with SoftMax is: −

In which y is output vector, and t is target label vector.

The simplicity of this function help to reduce amount of calculation to calculate the sensitive faster.

The function for computing sensitives of the network now as following. Differ is the subtraction of target to output vector.

## sensitive\_softmax.m

function [s] = sensitive\_softmax(differ,final\_output,final\_hidden2,final\_hidden

,weights) s{1,3}=[];

s{1,3} = -2\*differ;

s{1,2} = (final\_hidden2 .\* (1 final\_hidden2)).\*(weights{3,1}'\*s{1,3} ); s{1,1} = (final\_hidden .\* (1- final\_hidden)).\*(weights{2,1}'\*s{1,2} ); end

## Update forward function:

Since we’re using softmax instead of sigmoid for last layer, the forward function now is:

## forward.m

function [final\_hiddens,final\_output] = forward(inputs,hidden\_layers,weights\_array,biases\_array) final\_hiddens{1,hidden\_layers}=[];

% net output from hidden layer net\_hidden=inputs;

for i = 1 : hidden\_layers value\_hidden = weights\_array{i,1}\*

net\_hidden+biases\_array{i,1};

% final value from hidden layer after applying transfer function

tmp\_hidden =sigmoid(value\_hidden); final\_hiddens{1,i}=tmp\_hidden; net\_hidden=tmp\_hidden;

end

% net output from last layer

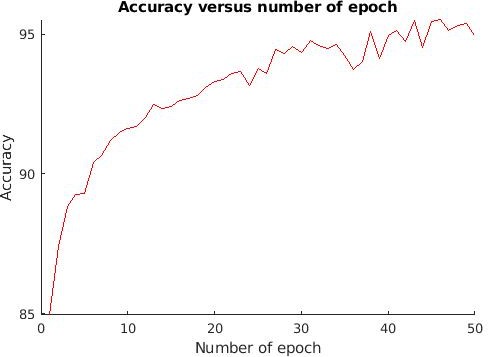
value\_final = weights\_array{1+hidden\_layers,1} \* net\_hidden + biases\_array{1+hidden\_layers,1};

% final value from last layer after applying transfer function final\_output = softmax(value\_final);

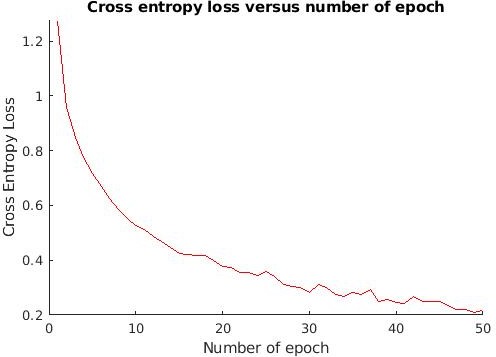
end

## Result of using SoftMax and cross entropy:

We train our network for 50 epochs. The graph of our training accuracy and cost is as follow.



*Figure 4: Graph of Accuracy versus number of epochs*

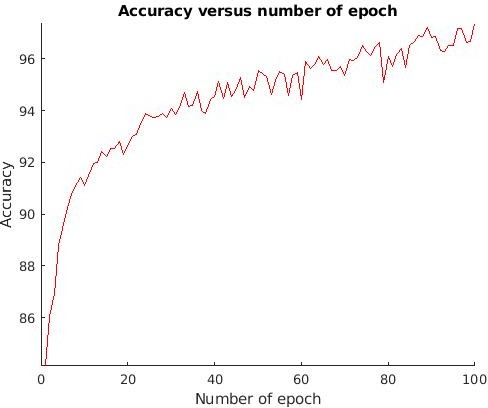


*Figure 5: Graph of Cross entropy loss versus number of epochs*

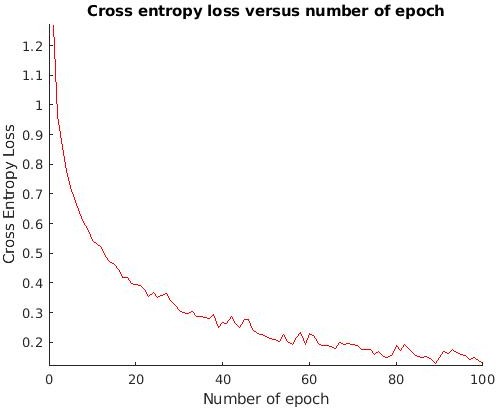
It looks like our network keeps reducing the cross-entropy loss and increase accuracy after each epoch. The accuracy after last epoch is 95.35, which is a good accuracy, and tends to reduce.

Then we can say that SoftMax and cross entropy bring a better performance and accuracy for our network. The training time is also slightly reduced, due to the amount and complexity of calculation of sensitives is reduce because of the derivative of cross entropy with SoftMax function.

We attempt to increase the number of epochs to 100 to see if our network still bring better accuracy and continue to reduce loss.



*Figure 6: Graph of Accuracy versus number of epochs after 100 epochs*



*Figure 7: Graph of Cross entropy loss versus number of epochs after 100 epochs*

From the results, we see that the accuracy was still increasing, and bring the final accuracy of our network is 97.37%. The cross entropy still keeps reducing and very close to zero.

When I use this network to test the test.csv data and upload to Kaggle challenge, the score is

0.89. It look likes there is a overfitting.

Then we can say that our SoftMax and cross entropy improved our network but has few signs of overfitting.

## Using mini batch gradient descent:

We attempt to train our network by dividing our training sets to equal size batches of example. The change of weights and biases will be accumulated and updated to the network’s weights and biases at the end of each epoch. We’re still using the final layer SoftMax and cross entropy loss function, because it’s bringing better result than sigmoid and reducing the amount of computing operation.

## Backpropagation minibatch algorithm

Similar to the first part, we need to generate random weights and biases, then do an iteration as the number of epochs for our training.

In each iteration, we will divide all training examples to multiple equal dimension mini batches and make a loop to go through all examples of a minibatches.

In one iteration of an example, we call a forward and backward pass to calculate output of each layers and calculate the change in weight and bias. But we do not update those changes to our network’s weights and biases in this iteration. We accumulate those changes and wait until the end of each epoch to update our network’s weight and bias.

## Implementation:

There are 2 function for minibatch update. One is **backProp\_Batch.m** to loop through all epochs and generate minibatches in each epoch. Another function is **backprop\_Step.m**, which goes through all examples of a minibatch to calculate the change in weight and bias to update at the end of an epoch.

## backProp\_Batch.m

function [weights,biases]= backProp\_Batch(m\_inputs,m\_targets,batch,input\_neurons,hidden\_neu rons,hidden\_neurons2,output\_neurons,epoch, learning\_rate)

% Number of sample data k = size(m\_inputs,1);

% To keep cost of the training arrayCost = zeros(epoch,1);

%To keep accuracy of the training arrayAcurracy = zeros(epoch,1);

% Number of batches num\_batch = k/batch;

% Network hyper parameters neurons=[input\_neurons,hidden\_neurons,hidden\_neurons2,output\_neu rons];

[weights,biases]=initWeights(2,neurons);

% Epoch iteration for i = 1: epoch

% Go through all mini batches for j = 1:num\_batch

offset = (j-1)\*batch+1; input=m\_inputs(offset:j\*batch,:); target = m\_targets(:,offset:j\*batch);

% Go through all example of a minibatch to keep the change in weights and biases [tmpweights,tmpbiases,cost]=backProp\_Step(input,target,2,neurons

,weights,biases, learning\_rate); weights{1,1}=weights{1,1}-tmpweights{1,1}; weights{2,1}=weights{2,1}-tmpweights{2,1}; weights{3,1}=weights{3,1}-tmpweights{3,1}; biases{1,1} = biases{1,1}- tmpbiases{1,1}; biases{2,1} = biases{2,1}- tmpbiases{2,1}; biases{3,1} = biases{3,1}- tmpbiases{3,1}; arrayCost(i)=arrayCost(i)+cost;

end

arrayCost(i) =arrayCost(i)/k; end

end

## backprop\_Step.m

function [dweights,dbiases,avgcost] = backProp\_Step(inputs,targets,hidden\_layers,neurons,weights,biase s, learning\_rate)

% Number of sample data k = size(inputs,1);

[dweights,dbiases]= emptyWeights(hidden\_layers,neurons);

% For each sample data sumcost = 0;

[mm,biasesmm]=emptyWeights(2,neurons);

for i = 1:k

% Feed forward [final\_hiddens,final\_output] =

forward\_Multi(inputs(i,:)',2,weights,biases);

% Calculate error

diff= (targets(:,i) - final\_output); cost=crossentropy\_loss(targets(:,i),final\_output); sumcost = sumcost + cost;

% Back-propagation start

% Calculate sensitive value for each layer s=sensitive\_softmax(diff,final\_output,final\_hiddens{1,2},final\_h iddens{1,1},weights);

% Accumulate weight changes

dweights{1,1} = dweights{1,1}+ learning\_rate .\* s{1,1}\* inputs(i,:);

dweights{2,1} = dweights{2,1}+ learning\_rate .\* s{1,2}\* final\_hiddens{1,1}';

dweights{3,1} = dweights{3,1}+ learning\_rate .\* s{1,3}\* final\_hiddens{1,2}';

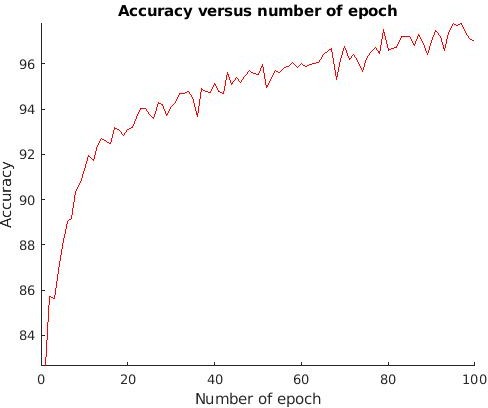
% Accumulate bias changes

dbiases{1,1} = dbiases{1,1}+learning\_rate \*s{1,1}; dbiases{2,1} = dbiases{2,1} +learning\_rate \*s{1,2}; dbiases{3,1}= dbiases{3,1} +learning\_rate \*s{1,3}; end

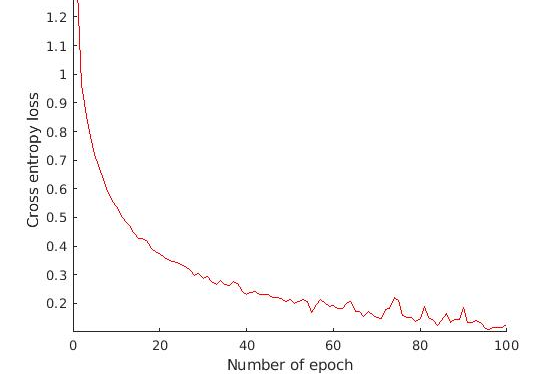
avgcost=sumcost; end

## Result of minibatch training

We use minibatch size of 25 to train our network, with the same hyper parameters as part 1.



*Figure 8: Graph of Accuracy versus number of epochs after 100 epochs*



*Figure 9: Graph of Cross entropy loss versus number of epochs after 100 epochs*

From the graphs, we see that the loss and accuracy are up and down and not stable than using SGD training, but it still tends to converge to smaller error, and increase accuracy after each epoch. The final accuracy was about 97%, which is good signal of improvement. The time of training is also slightly smaller than using SGD training.

## Using momentum to faster training

* 1. **Concept of momentum:**

By adding a term, like a velocity, which is related to the last previous change of weight can help the network to converge toward the small error quickly and help to train network faster. So, we attempt to implement this algorithm to our network to see if there are any improvement in speed of convergence.

## Implementation

We will add a team % = ∗ ∆%C( to each weight update. We will have a temporary variable to store current weight change to update to network weights for next epoch.

The code of backProp\_Batch. M is changed a little bit. The coefficient we choose is 0.5.

## backprop\_Batch.m

function [dweights,dbiases,avgcost] =

backProp\_Step(inputs,targets,hidden\_layers,neurons,weights,biase s, learning\_rate)

% Number of sample data k = size(inputs,1);

[dweights,dbiases]= emptyWeights(hidden\_layers,neurons);

% For each sample data sumcost = 0;

% To store current weight changes update to network weight in next epoch

[mm,biasesmm]=emptyWeights(2,neurons); for i = 1:k

% Feed forward [final\_hiddens,final\_output] =

forward\_Multi(inputs(i,:)',2,weights,biases);

% Calculate error

diff= (targets(:,i) - final\_output); cost=crossentropy\_loss(targets(:,i),final\_output); sumcost = sumcost + cost;

% Calculate sensitive value for each layer s=sensitive\_softmax(diff,final\_output,final\_hiddens{1,2},final\_h iddens{1,1},weights);

% Save change of weight matrices

delta1= learning\_rate .\* s{1,1}\* inputs(i,:); delta2= learning\_rate .\* s{1,2}\* final\_hiddens{1,1}'; delta3= learning\_rate .\* s{1,3}\* final\_hiddens{1,2}';

% Save change in weight with momentum term dweights{1,1} = dweights{1,1}+ delta1 + 0\*mm{1,1}; dweights{2,1} = dweights{2,1}+ delta2 + 0\*mm{2,1}; dweights{3,1} = dweights{3,1}+ delta3 + 0\*mm{3,1};

% Save change of weights for next momentum term mm{1,1}= delta1 + mm{1,1};

mm{2,1}= delta2 + mm{2,1};

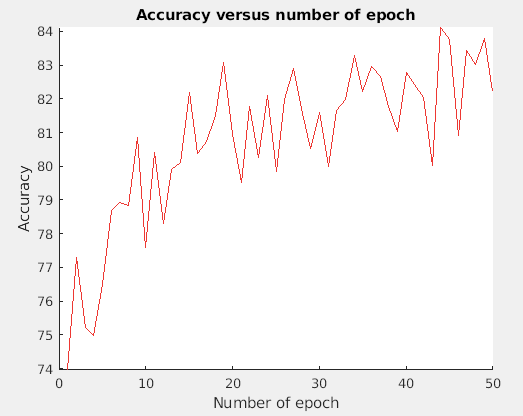
mm{3,1}= delta3 + mm{3,1};

% Save bias change

dbiases{1,1} = dbiases{1,1}+learning\_rate \*s{1,1}; dbiases{2,1} = dbiases{2,1} +learning\_rate \*s{1,2}; dbiases{3,1}= dbiases{3,1} +learning\_rate \*s{1,3};

end avgcost=sumcost; end

## Result



*Figure 10: Graph of Accuracy versus number of epochs after 50 epochs*

The accuracy looks like to go up and down too much and not stable. The final accuracy is also much lower than our previous attempt. So, we’re not choosing correct coefficient make the momentum improve the network. And our network are underfitting

# Conclusion

It is evident from this project and from the class as a whole that neural networks must be trained intelligently in order to produce meaningful results. Some training parameters that we found improved network performance include:

* Method of training: divide training to multiple batches each consider each example to update the weight. We notice that in minibatch, if we set the batch size is 1, the training process behave identically to Stochastic descent. And if we set the batch size equal to size of the training set, it behaves as batch gradient descent.
* Choosing activate function and loss function: by using SoftMax with cross entropy, we were able to reduce the amount of computation of sensitive, increase speed of training, and bring a better accuracy than using sigmoid with quadratic loss function.
* Normalize data: by normalize the data from our training set, we can make all input in range of [0, 1], with helps to reduce the complexity of calculation matrix, which is very expensive.
* One hot coding: by making the target label as vector of all zero and only one element is 1 help the calculation simpler (such as the derivative of cross entropy), and the network can perform better to converge with smaller error.
* Momentum: by adding a term that related to previous change of weight when updating network’s weights, we can speed up the training and make convergence faster. And we also consider potential of underfitting and not convergence

Things to consider for improving our network:

* Since our test result on Kaggle shows a sign of overfitting, we would like to attempt some solution for this issue, such as:
  + Cross validation: to separate training set to several portion and doing training along with validation to avoid bias.
  + L1 and L2 regulation.
  + Change dimension of our network.
  + And more …