Danny Ly

MNIST Report

Artificial Neural Network

# Introduction

The MNIST dataset consist of 60,000 various handwritten images, in a 28x28 pixel orientated vector space. This papers purpose is to discuss the researching findings of using an Artificial Neural network through training to recognize and help classify these images. The materials discussed in this document relate to University of Washington Artificial Neural network course. To explore further of how neural networks behave.

# The Task

The task to explore for this project is to understand the underlying mathematics as well as parameters and methods that can be applied to an artificial neural network to better help gain better classification performance. Aside from the main task, this project gives us a chance to further explore on our own the methodologies that were studied during the course work [1].

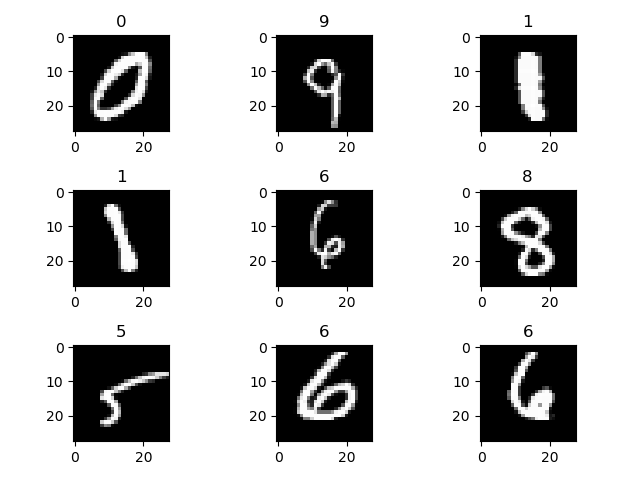


Figure Rendered digits

# Data Exploring

In this section we explore looking at the data in order to gain better insight on the problem at hand. The knowledge we would like to acquire from task are, what are the numbers of samples, what are the distribution of the data like?, are there clustering ? along with many methodologies that will allow us to help better choose our Artificial Neural network design.

## Statistics

Before starting this project, taking a look at the dataset that was given typically gives a good general idea of how to approach this problem. This knowledge that we use, can better help us decide what type of Neural network architecture would be possible in order to solve the problem.

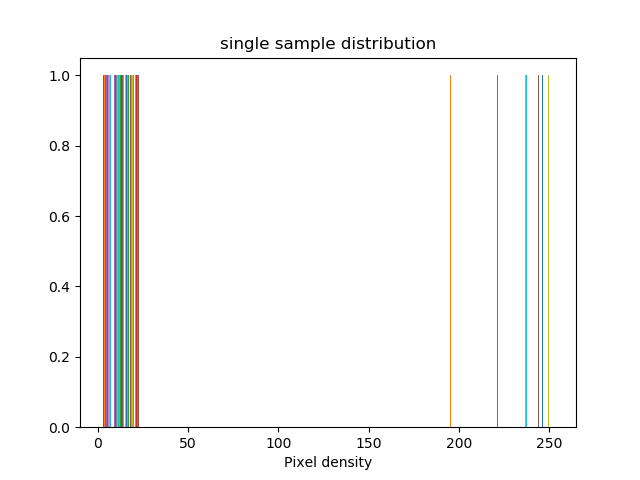


Figure Pixel Density for digit '5'

In the Above image I’ve plotted the distribution for a single sample, I did this instead of the entire dataset because, If we can understand a single sample and the problem. Each image is a vector in which each index holds a pixel density value between 0 and 255, calculating a mean in this case does not give us any more information of the entire population. Instead if we take a look at a single sample we can infer that each digit as we guess is of uniform distribution, meaning either a pixel will be of some gradient color or it will be black space. This tells us using method of normal distribution will not give us any more information about the dataset; not statistically significant.

## Dataset Distribution

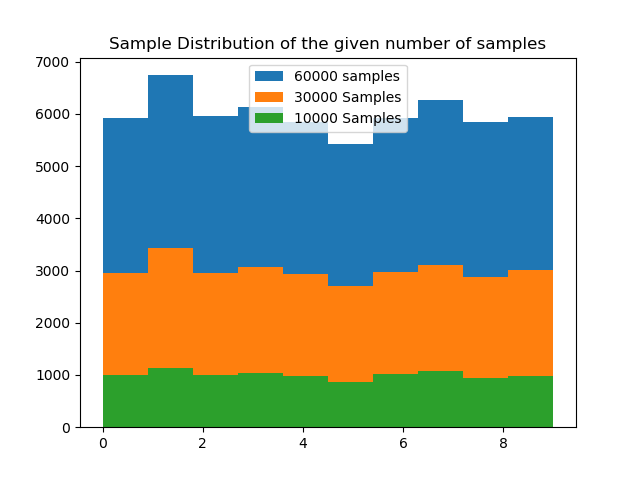


Figure Data Set Distribution

However here, we can start to think about the data, if there are 60,000 images, of this data we want to find out if there exist an even distribution of the sample. We do this because during training we do not want to introduce over dominate data to the network. If there were over dominate amounts of one observation over another we would therefore need to resample the population to get an even distribution. In our sample of 60,000 images, the above shows partitions of 10,000 and 30,000 as well. As we can see the distribution of the dataset is very even and normal, this tells us whether we train with 10 or 30 thousand samples there will be an even distribution of the data for our network in figure 3.

## PCA

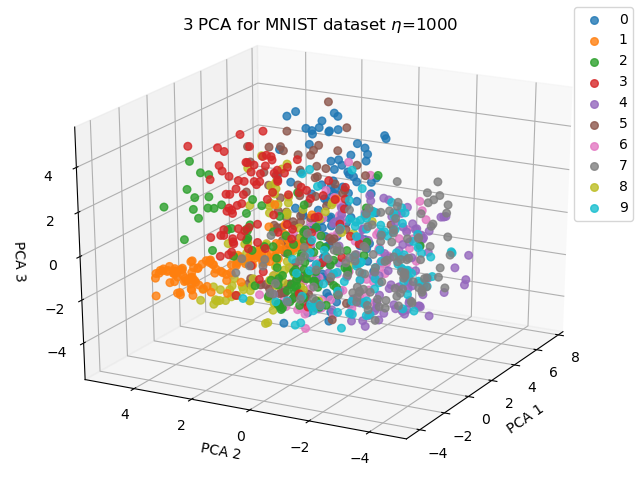
In this section we know our initial problem is to classify the digits, we also gathered data on the distribution of the dataset, and the samples within the dataset. Although we have numbers of such at our dispersal, making a visual representation of the dataset since it is well normalized in sample size to see what we are working with through PCA will be considered. 

Figure PCA for 1,000 samples

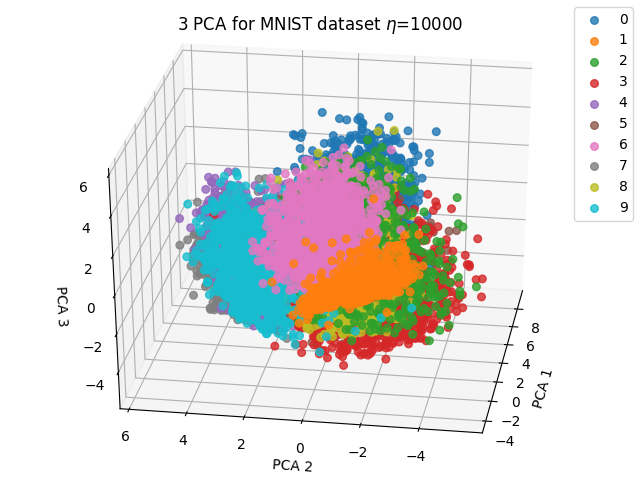
Performing PCA we get three features whose Eigen values are the highest amongst its neighbors, which means these features contain the most variance explained. In figure 4, we can now visually express there are potential forms of clusters.

Figure PCA for 10,000 Samples

Here we take more samples to see the mass of data definitely forms clusters. This is important knowledge to know since we can design an architecture comfortably knowing there exist strong correlations between the digits and much cluster is available.

# Design

Knowing the data we’ve gathered about, we can see through PCA that this dataset contains dense clustering. The design I decided to implement for this project is a **Competitive Neural Network**. I choose this network because of its ability to place multiple subclasses of neurons within the input space and slowly move towards the masses of their assigned classes. The main reason for this is, the visual representation of our data we can see large masses of clusters which pushes me to Competitive Neural Network. Also to note in figure 6 we explain how the neurons are able to move towards these clusters by the competitive learning rule.

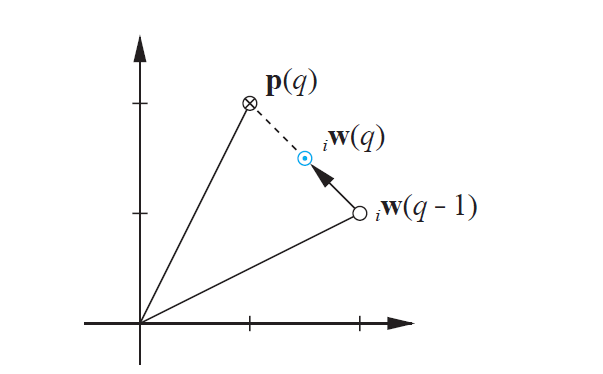


Figure Weights(q) moves closer to input p(q)

## Introduction

A Competitive Neural Network is a hybrid supervised learning network, this is because the first layer does not know the target outputs for the respective inputs until the second layer get the output from the first.

In a Competitive network the first layer exist of sub classes which are spread amongst the input vector spaces which are called prototype vector. When an input vector is introduced to the network each of these prototype vectors compete against one another, competition ends with whichever neuron’s prototype vector is closest to the input vector presented. Once the winning Neuron has been chosen its Weights will move closer to the input vector space defined by Kohonen learning rule:

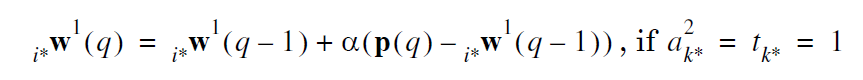


Figure Kohonen Learning Rule

While there exist a few versions of this learning rule, that will be discussed further in methodologies .

## Architecture

For this networks design I went with a 2 layer network, the first network consist of a competitive layer and the second linear activation function.

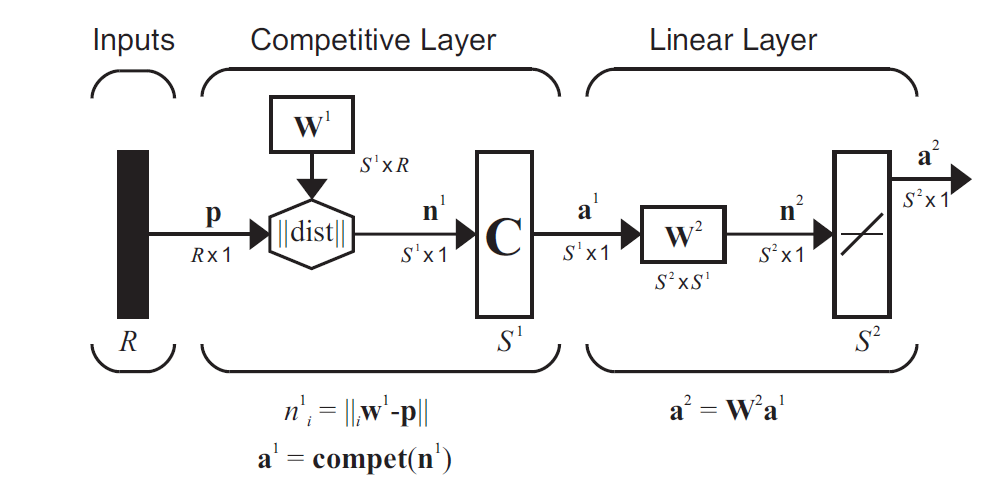


Figure [1] Architecture Design

In this design I used 150 neurons to form the subclasses in the competitive layer with 10 final neurons for the output layer. The second layer is necessarily since after each neuron in the first class has developed their individual prototype vectors there needs to be a way to combine all subclasses into their respective classes of the 10 digits. This is where the weights of the 2nd layer come in. The weights in the second layer combines the output of the first later into final classes, this is done through grouping each subclass neuron to their final output classes. For example in this design there exist 150 neuron in the first layer and 10 final neurons for the final layer. Dividing the final layer by the sub layers we have 10 neurons per class.

## Specifications:

= 150

R = 784

= 150 X 784

= -||||

= Compet()

= 10 X

= X1

**Input:** 1X784 Vector *non-normalized*

**Competitive Transfer Function:** in the first layer the competitive transfer function works by looking for the largest ecludiean distance within each subclass, this distance explains how close a prototype vector is from the input vector

**Pure Linear Transfer Function:**  In the second layer the purlin function is used to combine all the neurons in the first layer to their respective class layers

**Arguments:**

* *Epochs : integer, default= 5*
* *Learning\_rate : float, default=.1*
* *Verbose: Boolean, default = False*
* *DataDrivenInitalization: Boolean, default= True*
* *Conscience: Boolean, default = False*

# Methodologies

In this section I speak more in-depth s of my findings and design, as well as how I was able to solve them and any other methodologies that might have impacted my performance/ results. Here I layout each steps I took to design this network and within each step the things I discovered.

## Step 1 – Initial Layer 1 weights:

One of the most important aspect of this competitive network is that the initial prototype neurons in the first layer need to be dispersed within the input space, the few probvlems that arise in competitive neurons are neurons that do not cover the input space, where data lies, or neurons that are in input spaces where no data lie, these are considered *dead neurons.*

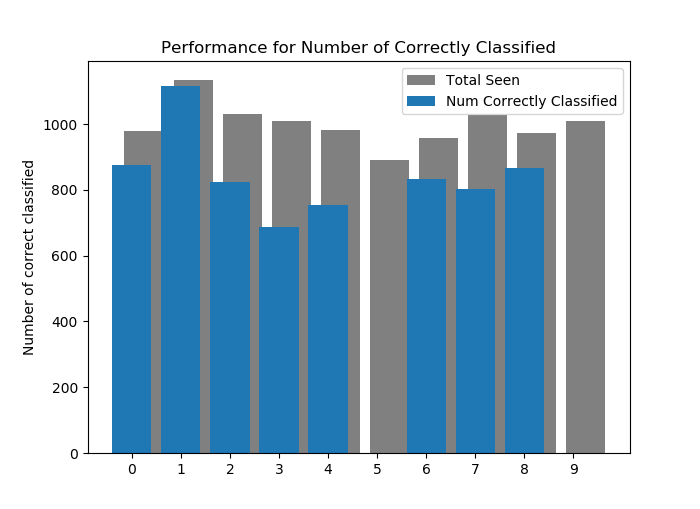
Initially for This design I decided to go with prototype vectors of uniform randomized picking. However I ran into problems where specific neurons were scattered within a cluster that were interfering with other neurons region and would be pushed away from where they needed to be.

Figure Performance Of network with uniform weights

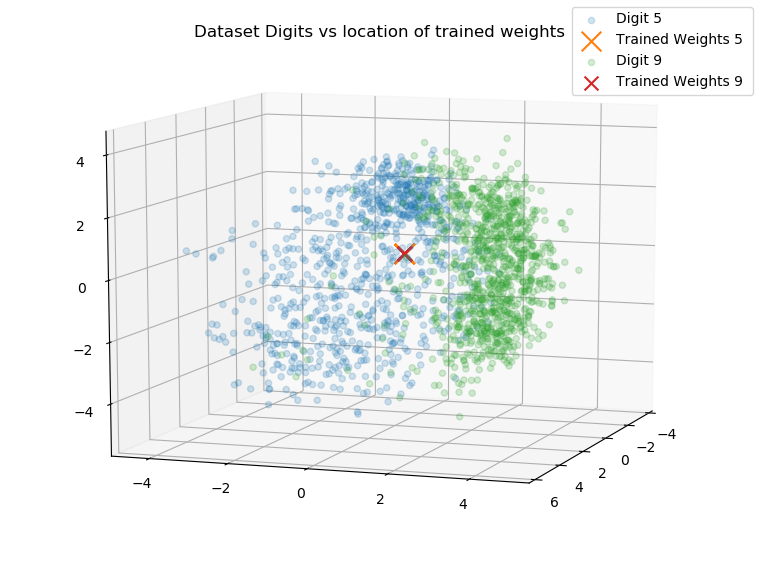
The result would be Neurons never being able to reach the input vectors they were assign to, as we can see from figure 8, the performance for this network did not recognize input vector 5 or 9. Upon further inspection

Figure 'Stacking' of initial Weights

As seen in figure 9, there are two neurons which are neurons in the cluster that belong to digit 9 and digit 5, as we can see the input space around the area between them, the problem here is the initialization of the random uniform weights put them dead in the center of these two classes, which meant when the competitive function looked for a winner these weights would oscillate between territories, therefore we have a stack of neurons in the center of the two input patterns.

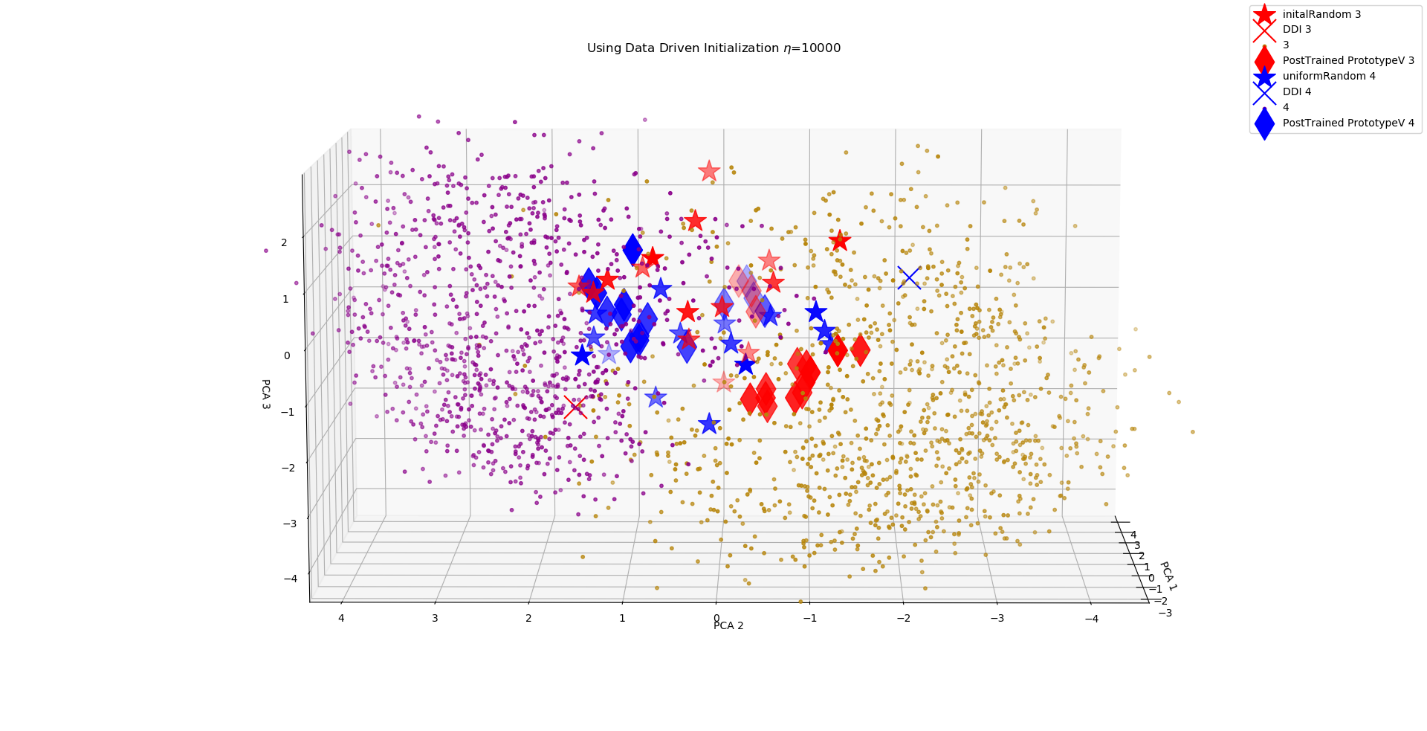
To solve this issue there lies a method called Data Driven Initialization where each prototype neuron randomly chooses a input from the input vectors as their initial weights, this type of initialization allows each of the subclasses a better starting point to be within the range of their respective masses.

Figure Using Data Driven Initialization

As we can see above the ‘X’’s mark where a **single**  randomly selected input vector would be for a neuron’s initialized weights, the red ‘X’ on the bottom left corner represents the digit 3’s and the blue ‘X’ represented the digit 4. The Stars in the figure show the weights of random uniform picking. The important thing to notice here is that the single selected random data driven initialized weights are a lot more dispersed within their respective clusters, whereas the uniform random weights will likely cause a ‘stacking’ since both the initial weights invade territories of opposite classes.

To do a data driven initialization, randomly pick input vectors from the dataset.

## Step 2 – Layer 2 Weights

To select 2nd layer weights, we first must choose the number of neurons within the first layer, this is because the second layers weights is a union of the first layers target vectors, the problem I ran into here is for each input vector the target labels were a integer value. In order to convert those into 10x1 target vectors to match the final layers output of a^2 I used a method called *one-hot-encoding*  which allowed me to convert normal integers to unique vectors that represented them.

To find the orientation of the weights in the second layer use:

This means that for each class there exist a ‘neuron per group’ for that class, this is used to combine the neurons in the first layer to the final output layer. These groups are then stacked against each other to produce a place for each neuron in the first layer to classify to their respective classes in the second layer.

## Step 3 – Object Design:

After the weights have been selected, this design uses a array list to replicate holding of each layer, for each index exist a layer in the network for this network we use index 0, and 1. These index’s represent their respective layers, for each layer the array list holds a hash table with 3 key elements

1. ‘Weights’ : Neurons X input Vector size : matrix
2. ‘Transfunction’ : a respective Transfer function
3. ‘Bias’ : respective bias

When the network is first called, e initial the first and second layers, with a data driven initialization approach. After the network has been set up a call to the function train() takes 2 parameters a input matrix of the dataset and its respective target vector.

During training the network first takes the nth input vector and calculates the Euclidean distance between each prototype vector and the current input vector.

Once the winning Neuron is selected, we must check whether a classification was correctly made, if there was a correct classification then we update that winning neurons weights. However if there was a false positive winning neuron we must move this false positive neuron away from this input vector and move the second closest neuron towards the input vector.

## Step 4—Learning rate:

One of the most important hyper parameters of this network is the learning rate, because with a competitive layer the winning neuron moves closer to the input vector, however the learning rate is what decides the magnitude of this move.

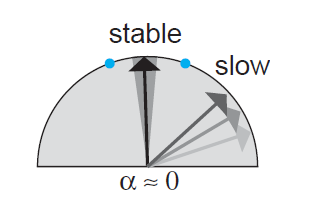


Figure [1] a low learning rate

As seen in figure 12. A low learning rate allows the network to converge between the masses in small steps, the down side to this is learning will be much slower since the prototype vectors must move to the respective locations.

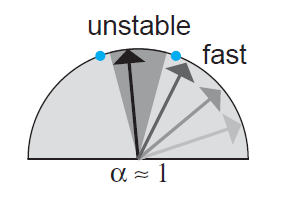


Figure [1] Fast Learning rate

However, a high learning rate will help learning faster but be quite unstable since the neuron will esstentially jump between input vectors it is introduced to.

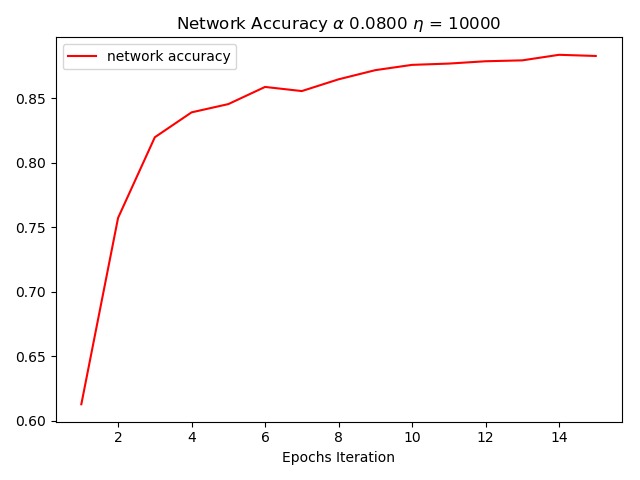
At first I approached this problem with trying to find a mean balance of a equilibrium of a learning rate, that would fit a mediuam of low and high learning rate to grab botrh stable and fast learning. Thje results seen below in figure 14 show promise as the network slowly converges 

Figure Stable static Learning rate

In figure 14 shows a trained sample size of 30,000 and a static α = .08 for the accuracy we get about 91% correct classification

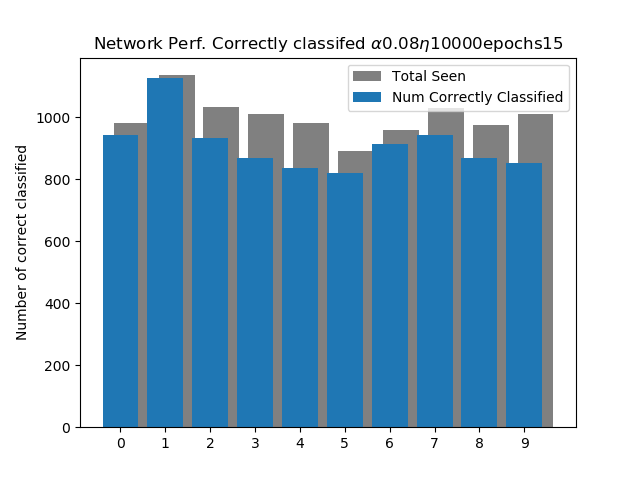


Figure Performance on test set

The respective Test set performance is showed in figure 15, as we can see given new data to the network, the network performs favorably. However over time of running multiple different configurations using more samples, less samples or a little bit more neurons in the hidden layer, the performance would always plateau at about 91% accuracy(.8% error)

This thinking behind this could be the culprit that was not being adjusted the **learning rate.** Understanding the network’s learning rule helped discover why the performance would plateau at about 90%, one favorable explanation was that the prototype vectors during training had converged to a location within the mass with a couple of input vectors that left out a few of the distorted inputs.

With this in mind, one way to test this theory was to implement an **dynamic learning rate**, here a dynamic learning rate is one that reads each iterations error, and compares it to the pervious performance. If the error is starting to decrease this tells us that the network is classifying correctly. The next step would be to check whether the performance is stabilizing or still shifting around.

If the network is slowly stabilizing we decrease out current learning rate. Decreasing the learning rate will help the neurons slowly converge to the center of mass which solves it moving to a specific location within the mass.

As the neuron’s learning rate is decrease so does its attraction to new input vectors not seen, this can be thought of as a way to ‘generalize’. If the error performance starts to increase we will shift out learning rate up , which can be thought of as , fixing the mistake we made last, since the last decrease in learning rate resulted in the lowest possible.

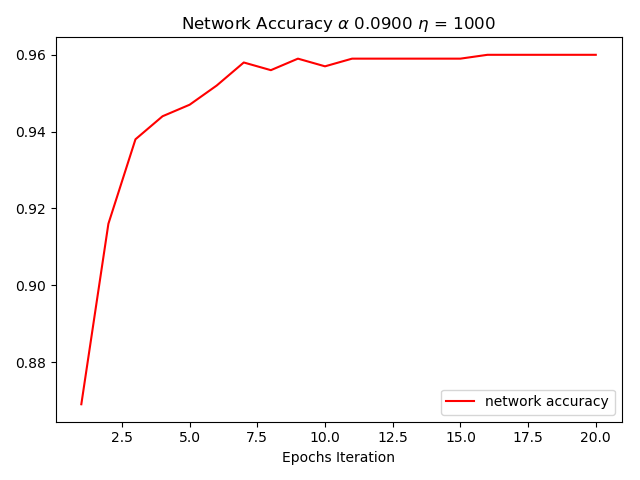
Implementing this dynamic learning rate proved to be hypothetically correct, since our results for the error now decreased in 4 % from 91% accuracy to 95% accuracy. In the below images we can see the following performance metrics of the networking using a dynamic learning rate, just as the static leraning rate we can see the network have rough edges in the beginning however start to smoothen out towards the end. The explanation for this is in the beginning our learning rate is large so the network is able to jump around and capture the large mass it is attracted to, then as the error becomes smaller the learning rate is lowered such that new input vectors do not pull it around, which explains the smooth ending curve towards the end of training.

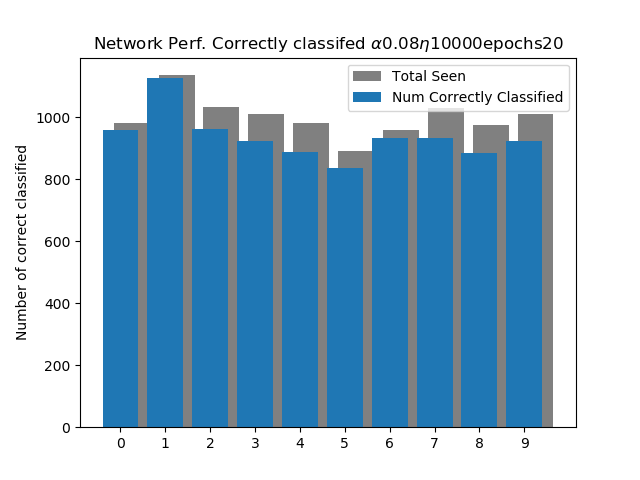
Figure Dynamic Learning Rate

Figure Testing Set Dynamic Learning Rate

In Figure 17 we see the performance of the test set used on the trained network with a dynamic learning rate, although these shapes are quite similar to the static learning rate we can take a closer look and see that for digits 9 and 5 more of them clusters are being captured here with a dynamic learning rule.

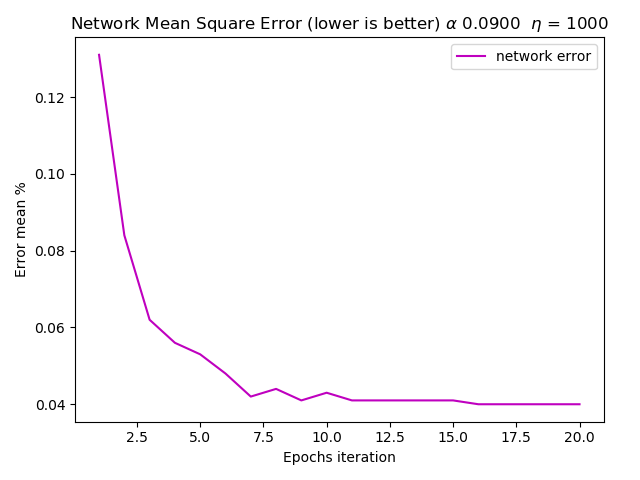


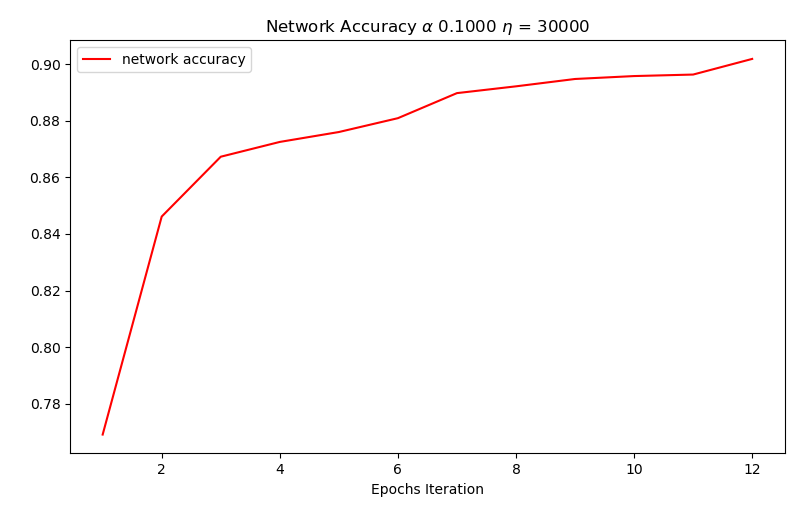
Figure Error of Dynamic Learning Rate

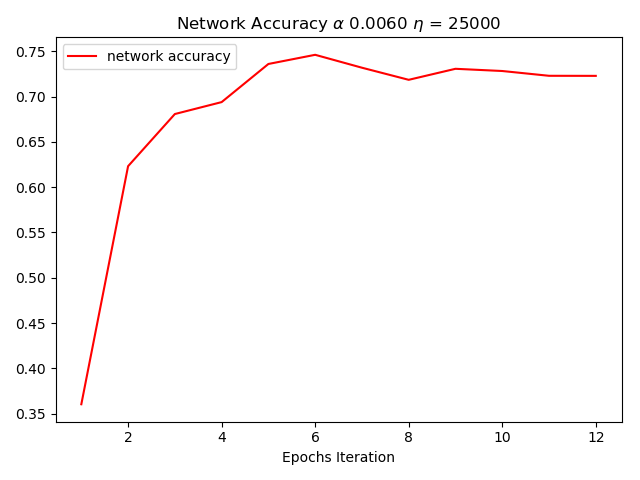
## Problems:

*Normalization -* After finishing the network, testing the network showed very poor performance, After closer inspection the Learning Vector Quantization works best when the input vectors are not normalized. This is also one of the advantages of using an LVQ neural network. The reason for an improved learning rate was because the when the input vectors were normalized the distances between the prototype vectors and input vectors were all clustered together, and would make it very difficult for neurons to migrate to where they needed to go.

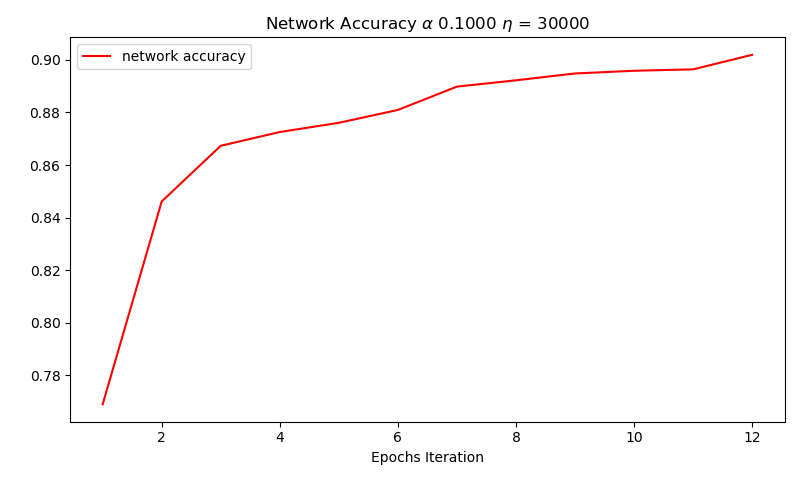
*Adaptive bias –* ***‘****conscience’ s*  as the text states it is when we penalize all neurons when a winning neuron is chosen, this mixes’ all neurons whom have not had a chance to win to win, this is because the formula for implementing conscience is to multiply all neurons bias by .09 while subtracting -.02 to the winning neuron. However this does not implement well with a data driven Initialization of the weights, because we choose to initalize the weights by random input vectors each vector in theory should never be dead and within or near the mass of the input clusters. Which resulted in worst lower performance since we did not see any dead neurons, the root explanation for this was because of such low amount of neurons in the first competitive layer they did not have much neurons to compete with.

Results

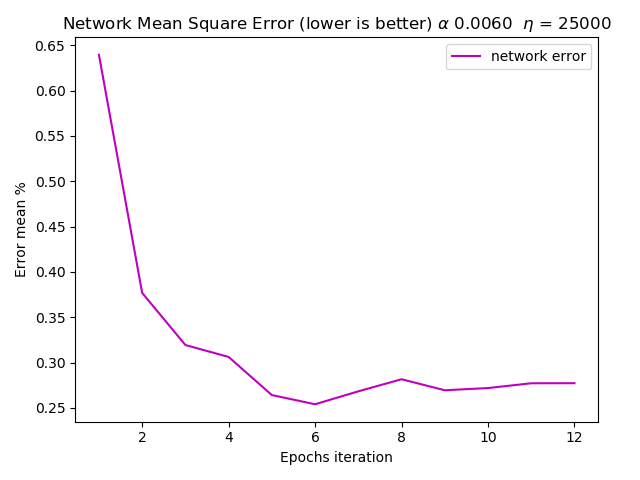


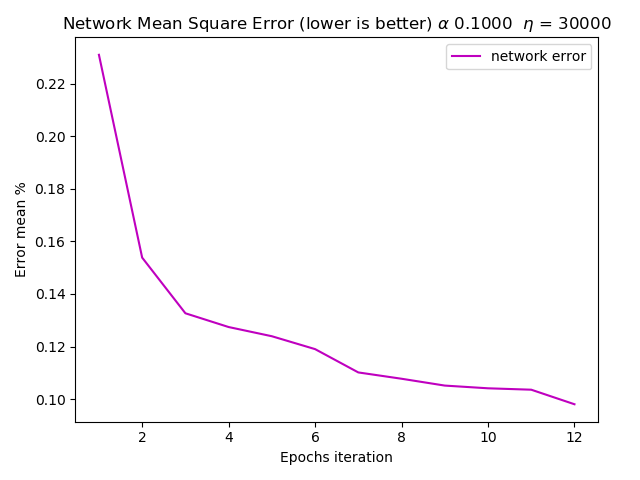
See the above, my findings for using the ecludian distane without noralizing the inputs give better results this can be a guess of the input spaces magnitude that is lost, such that the book also disscuess that there is not NEED to normalize input for an LVQ2 network, as we can see also here that with a smaller learning rate we also can see a steadier leraning curve for the network, this is also explained by the depth of steps we take for each correct classification see below

Above we can see with a much lower learning rate we get a slower performing network, we also can notice the very edgy learning curve, this can be explained by the neurons moving at a much slower, step, therefore we et neurons who have not completely learned patterns since they learn much slower

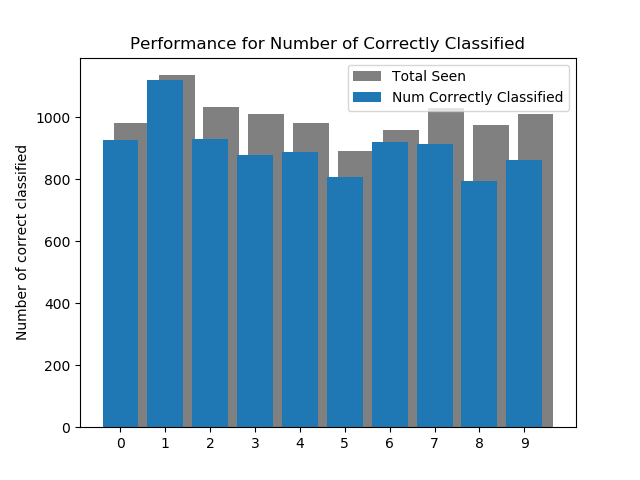


In the above image , notice that with this much higher learning rate the leraning curve is much smoother, this is expected since the higher/larger steps the winning neuron is allowed to take towards it correctly classified input vector the faster it will be able to correctly classify that patterns further down the more epochs we take.

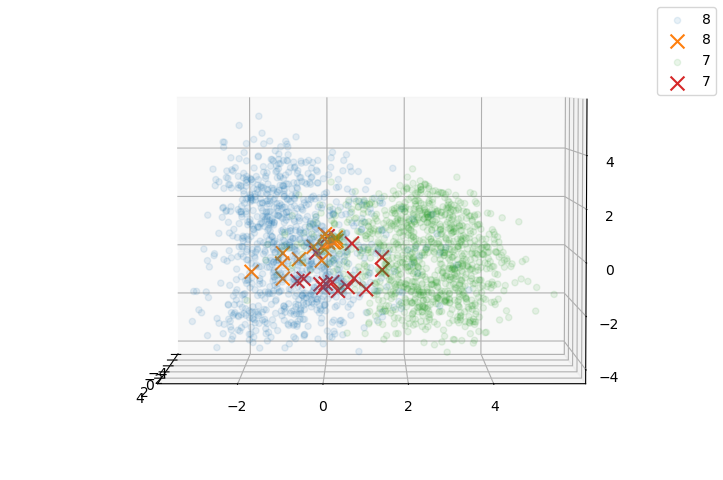


In the above we can see the respective error rate for this configuration, its clear that the error of the network will a much smaller learning rate will be ‘rough’ in shape, this is explained by the small steps we take

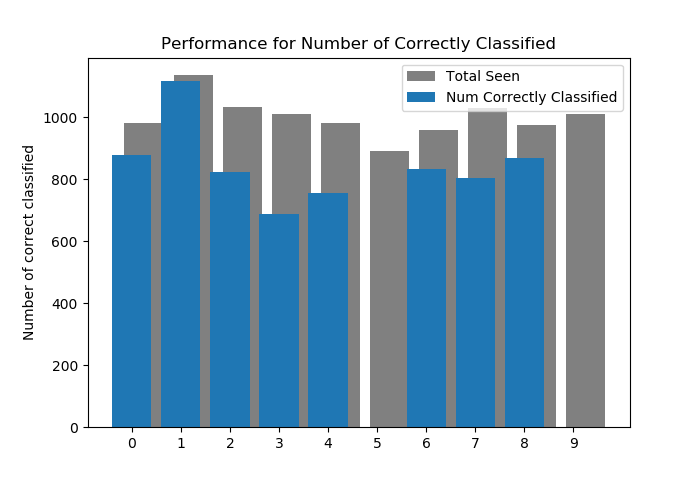
In the above image we have adjusted our learning rate higher, as well as given the network more patterns to see, as you can see the error curve is much smoother, this is explained through the learning curve since the network is allowed a larger step towards or away from their respective patterns, each epoch we can expect the patterns to be classified more correctly. Giving this network more epochs will theorthically produce a very smooth curve.

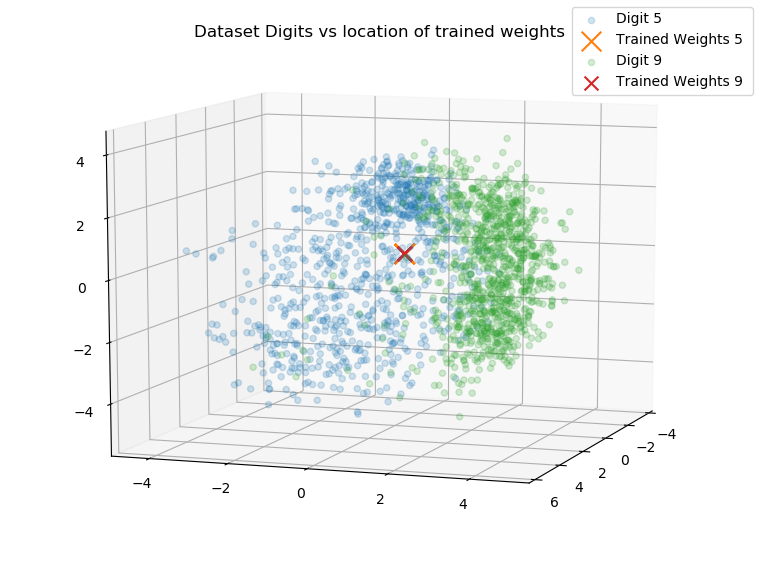
Methods 

Above we can see how I measured the performance using the testing dataset , which consist of 10000 samples, we can see a pretty good average performance

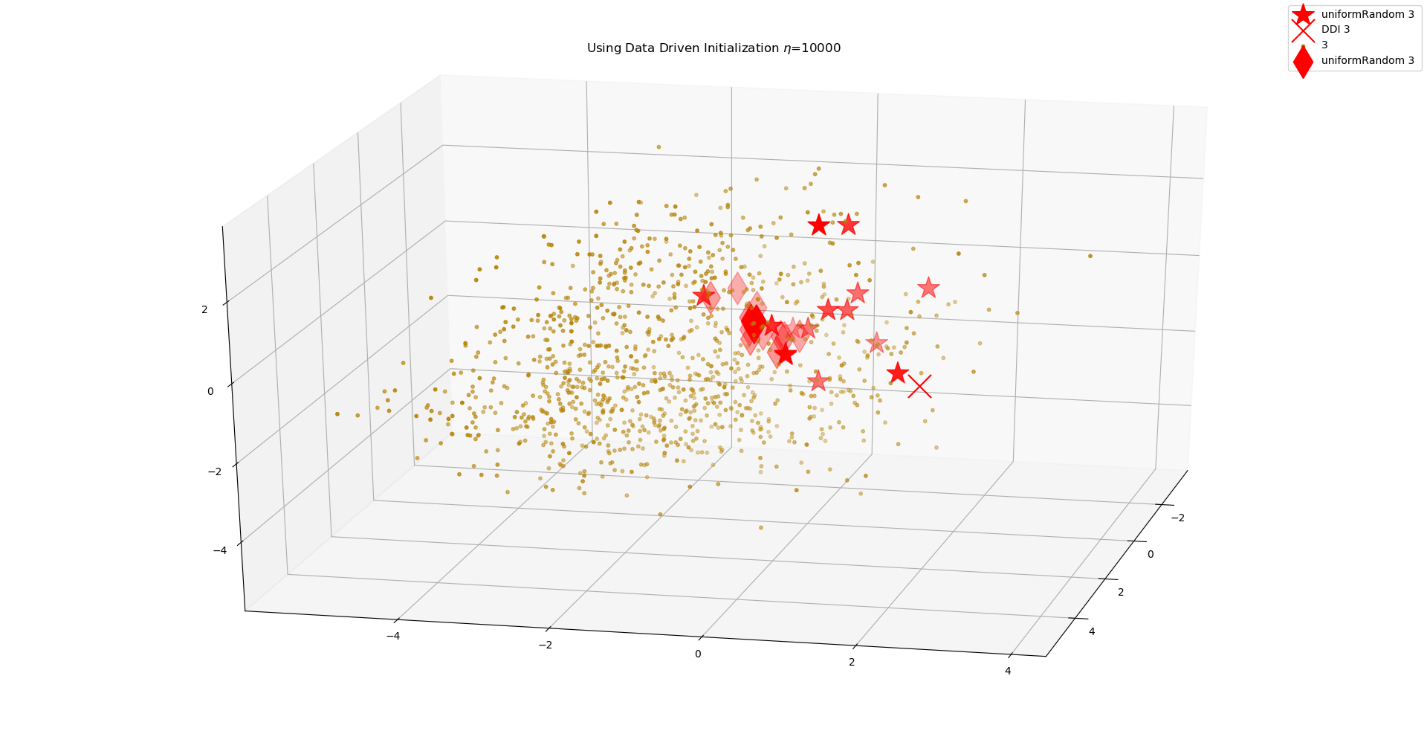
because I wanted to visualize the weights after traininig and how they are placed within the input space I was able to performe the same analysis on the weights as the input space, here we see a sample of digits 7 and 8 

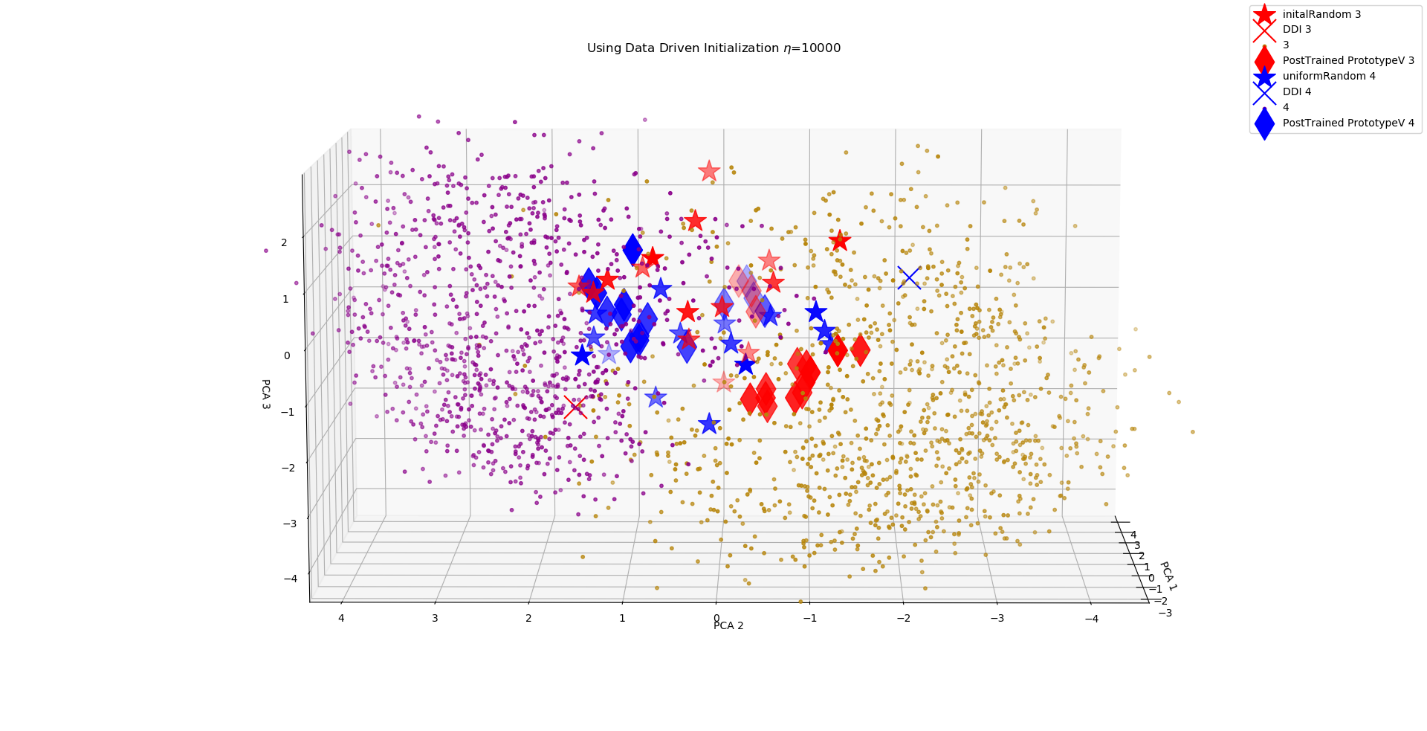
Here we can see that the weights although are still close, have a solid respective separation between the two cluster masses, this is a good sign since our training just introduced 25000 input data on 12 epoach that hypothesis is more training with a dynamic decaying learning rate will give us better results.



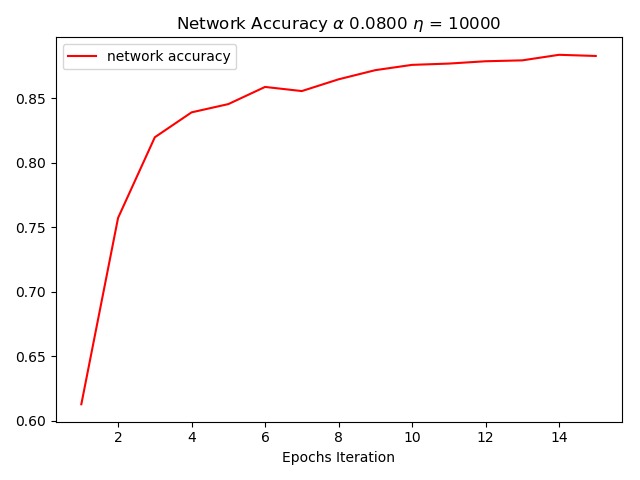
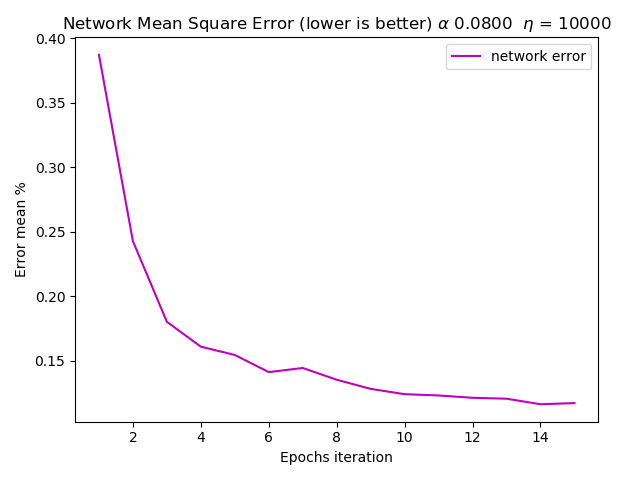
Through some error an trials the above shows the performance of the network over a dataset its never seen before, which contain 10,000 samples, here our configureation for the network is n=30K, alpha=.006, epoch = 12. As we can see something strange is happending the network seems to almost not know the existence of learn what digits 5 and 9 are. Upon further inspection we can see that the prototype vectors for digits 8 and 9 are almost stacked onto of Aeach other, this might be exaplined through the initaliozatioin of our weights, the Kohnene rule for the leanring vector quanilization formula says that false postivie neurons move away from the input and runne ups move closer, however if we have a glob of initial prototype vectors that are very close to eachother, wether they win or not, they will continue to oscilate between where they are suppoused to go and back away from the input of their neighbors Look closly we can see a stacking of the prototype vectors for 5 and 9where they are wedge between where they are suppoused to go, as stated inm the NNDesign book one of the problems of an LVQ2 network are if a initial weight has to travel through a reigon of a class that it doesn’;t represent to get to where it need to go it will be repulsed by vectors in that tegion(see book page 636/1012)

The learning rate is very sensitive



In the above image we have a 

In the above figure we can see the idamonds representing the Post trained prototype vectors , are behaving correctly, the stars are where we are suing a random uniform number generator for our initial weights and the ‘X’ markers are if we were to use data driven initialization of the weights, this does confirm out network is performing as it should however, notice that the stars for the uniform random generalization are more or less scatter all about the center of the input space.



Ideas: implement a dynamic adjusting learning rate, when the error rate seems to be going high or stabilizing we will start to decay the learning rate, however if the network still have a error rate lets step up the learning rate

Sources

<https://stackoverflow.com/questions/40427435/extract-images-from-idx3-ubyte-file-or-gzip-via-python> :directions to parse the mnist data file

<https://github.com/sorki/python-mnist/blob/master/mnist/loader.py> : python-mnist loader, replace . with – since in his code, that’s the syntax

https://matplotlib.org/tutorials/intermediate/tight\_layout\_guide.html : plotting guide

<http://alexanderfabisch.github.io/t-sne-in-scikit-learn.html> : TSNE

[1] Neural Network Design 2nd. Editon Martin TR. Hagen,