Multi-Layer Competitive Neural Networks for Image Classification

Artificial Neural Networks

Danny Ly

Technical report

December 2017

Abstract

There exist many methods of pattern recognition with various performances index and technique, one technique however that is not very much explored is the use of Competitive Network in Artificial Neural Networks sub class. In this article we further explore implementation of the Competitive Neural network architecture and asses its performance to learn and recognize the MNIST hand written dataset. It turns out the learning vector quantization 2 learning rule works very well in pattern recognition, this network is able to achieve 96% accuracy with non-normalized input patterns. With the exploration of methods such as Data Driven Initialization, adaptive bias and dynamic learning rate this network was able to perform within competitive ranges.

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].

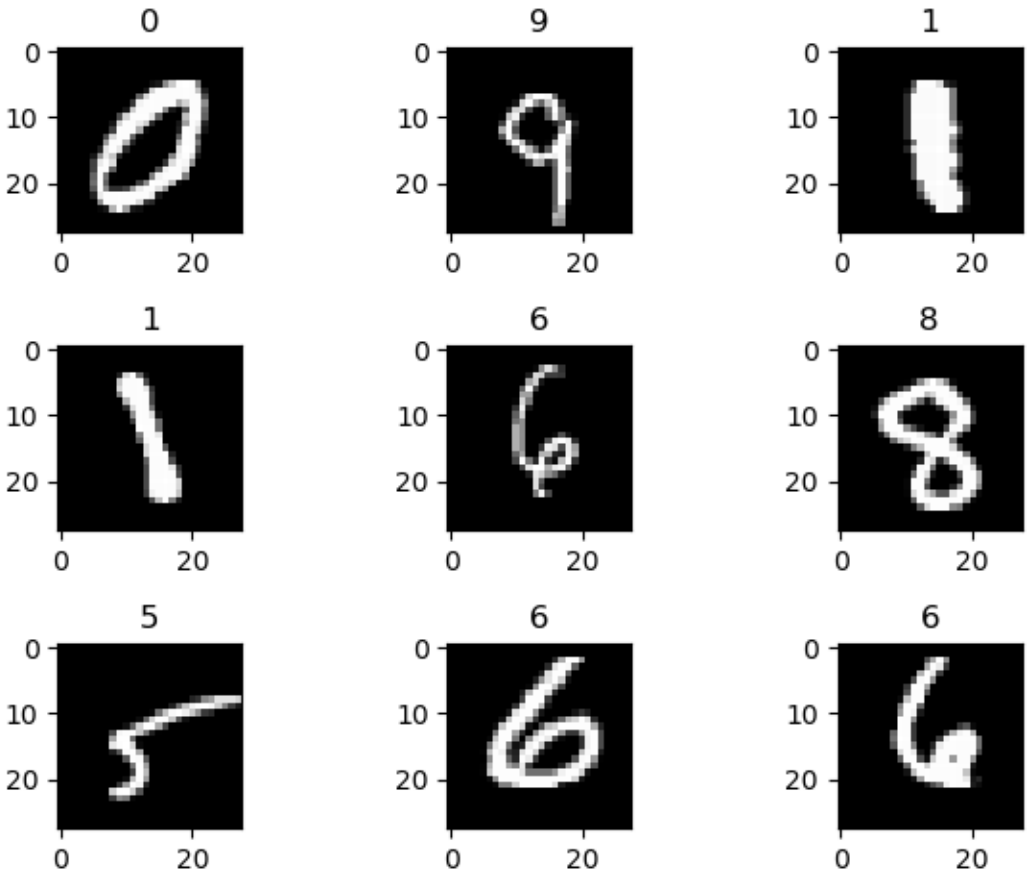


Figure 1 Rendered digits

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.

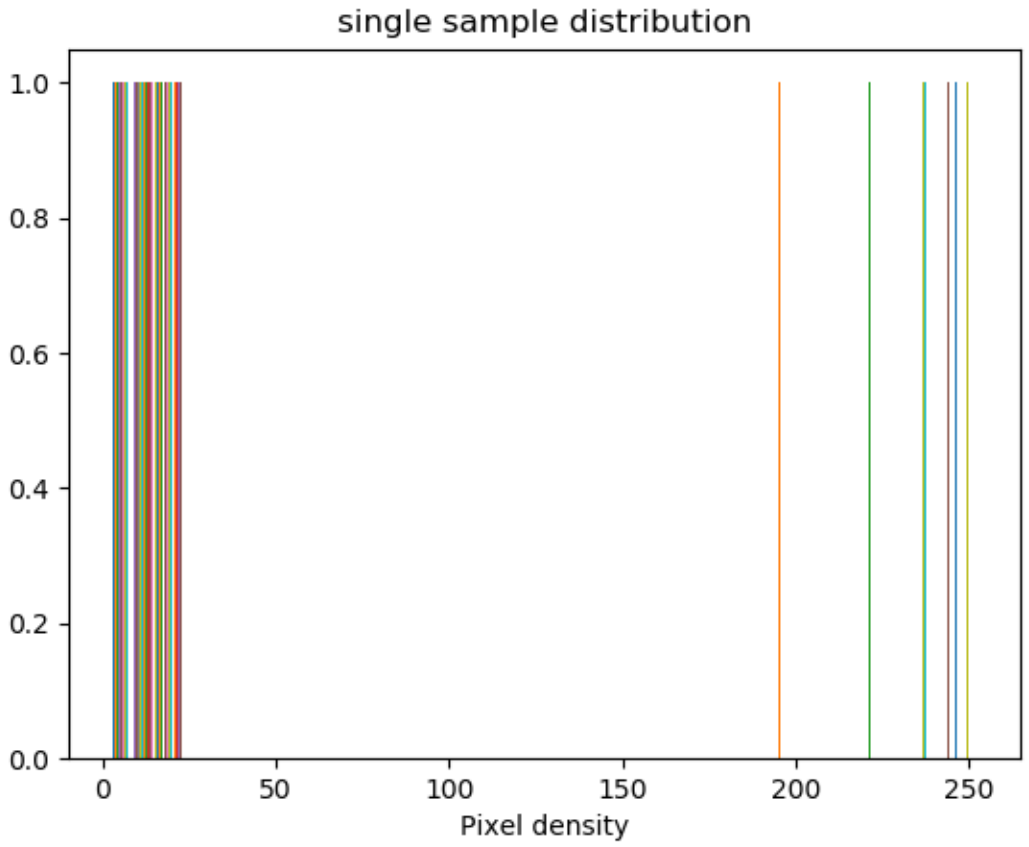


Figure 2 Pixel Density for digit '5'

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

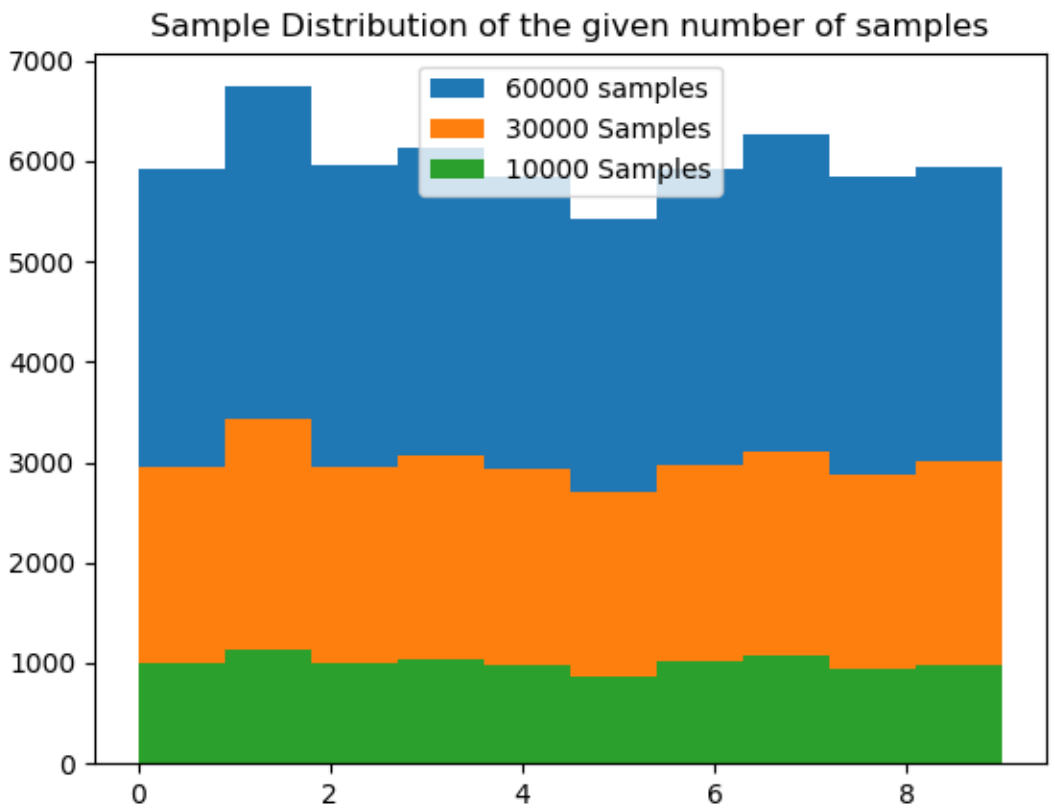


Figure 3 Data Set Distribution

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.

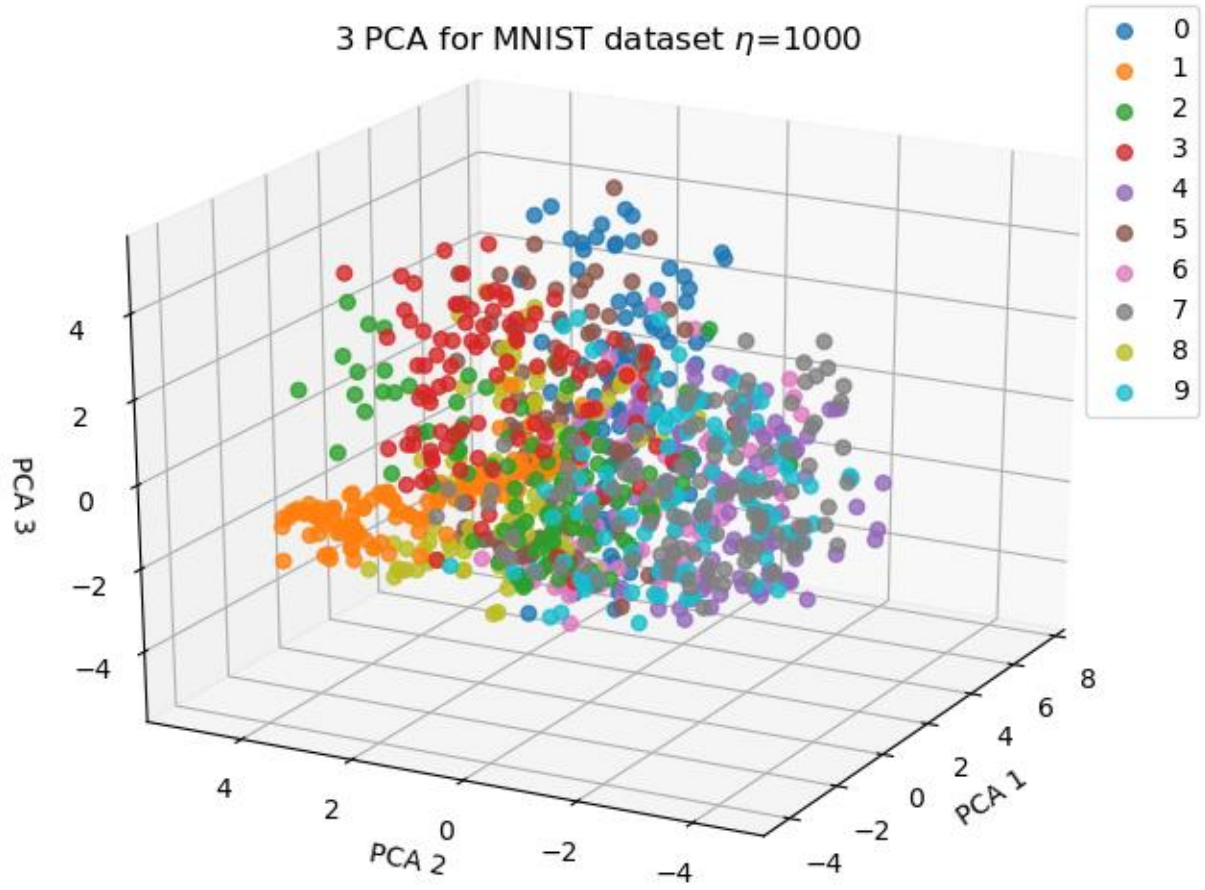


Figure 4 PCA for 1,000 samples

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.

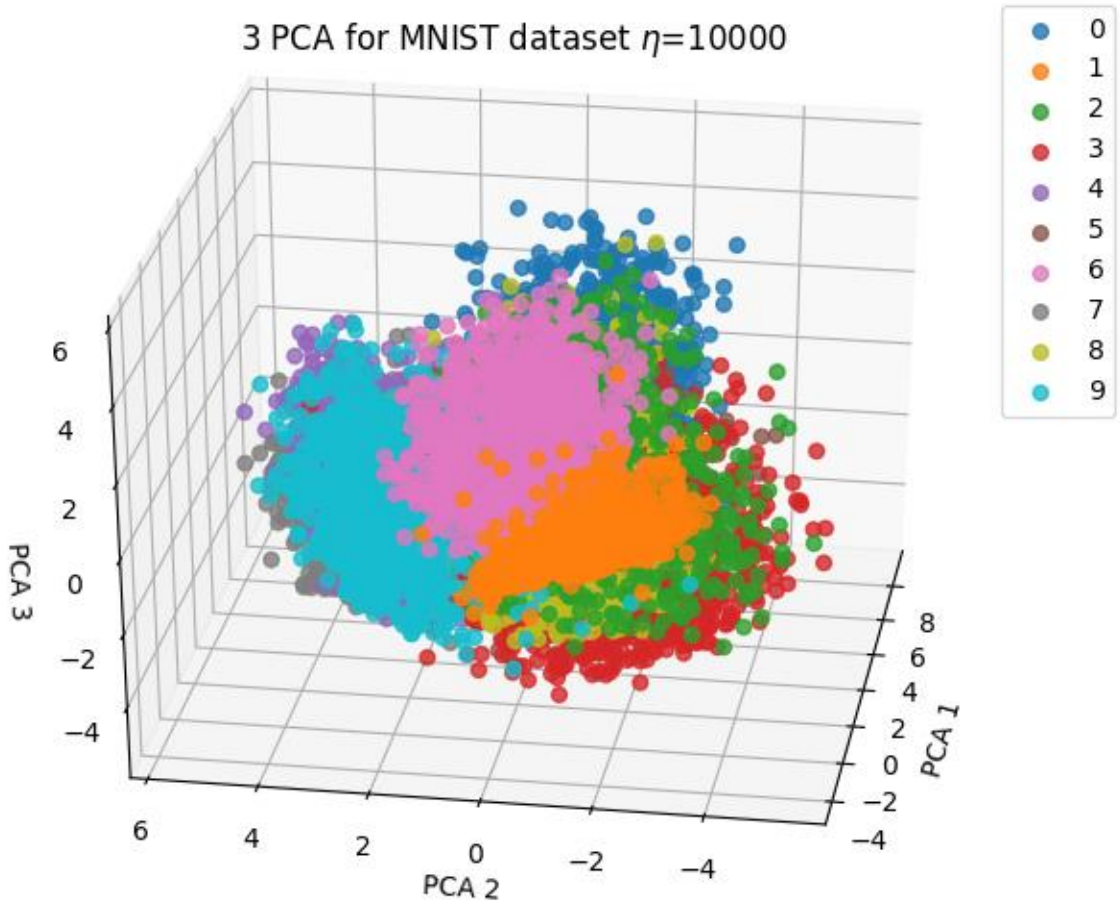


Figure 5 PCA for 10,000 Samples

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.

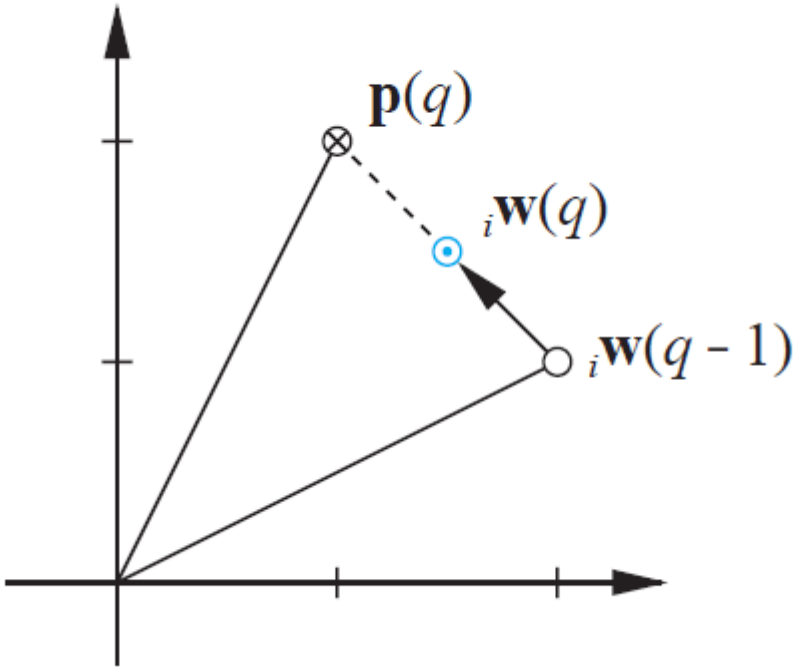


Figure 6 Weights (q) moves closer to input p (q)

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:

$$ {}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a^2_{k^*} = t_{k^*} = 1 $$

Figure 7 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.

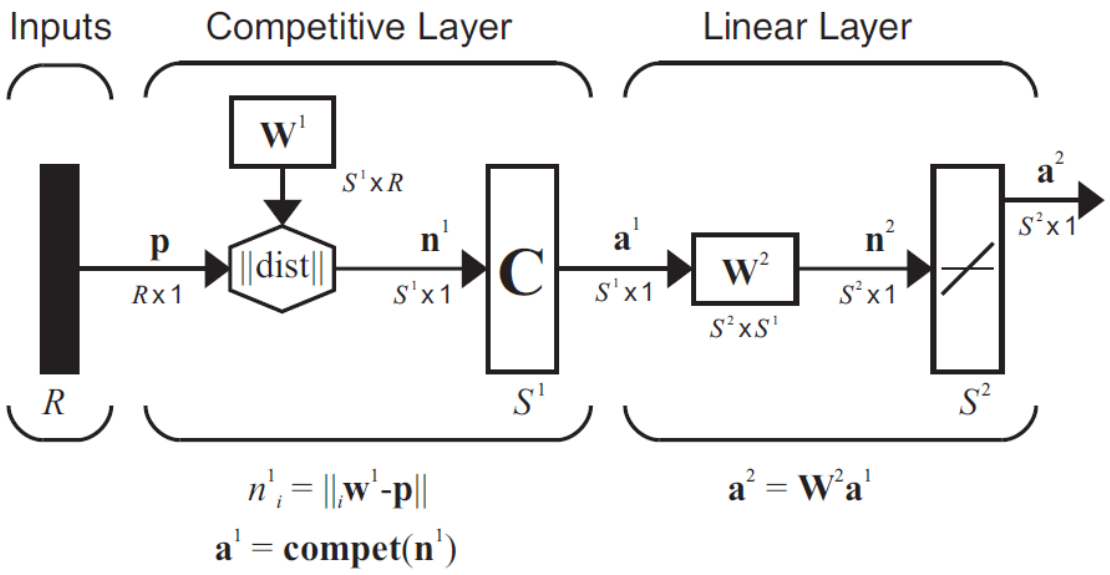


Figure 8 [1] Architecture Design

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:

S^1 = 150

R = 784

W^1= 150 X 784

N^1 = -||$W^{1^T} - P$||

a^1= Compet (N^1)

W^2= 10 X S^1

a^2 = S^2X1

Input: 1X784 Vector *non-normalized*

Competitive Transfer Function: in the first layer the competitive transfer function works by looking for the largest Euclidean 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 problems 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.

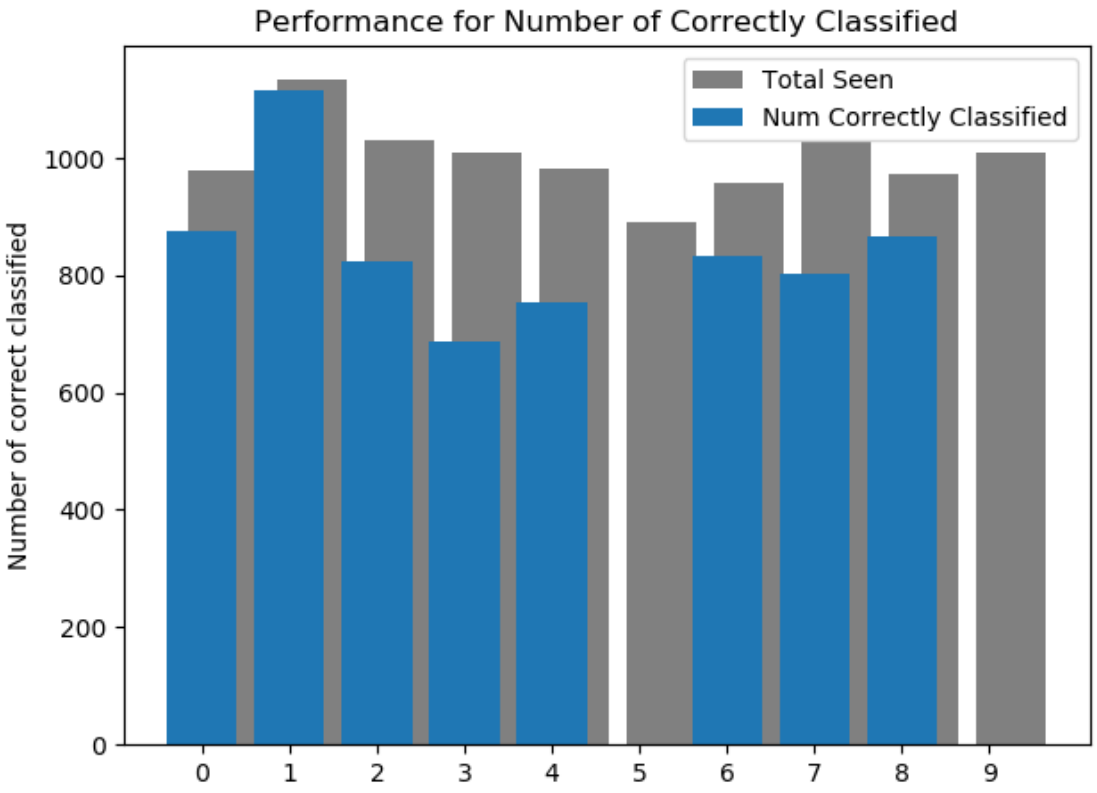


Figure 9 Performance Of network with uniform weights

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

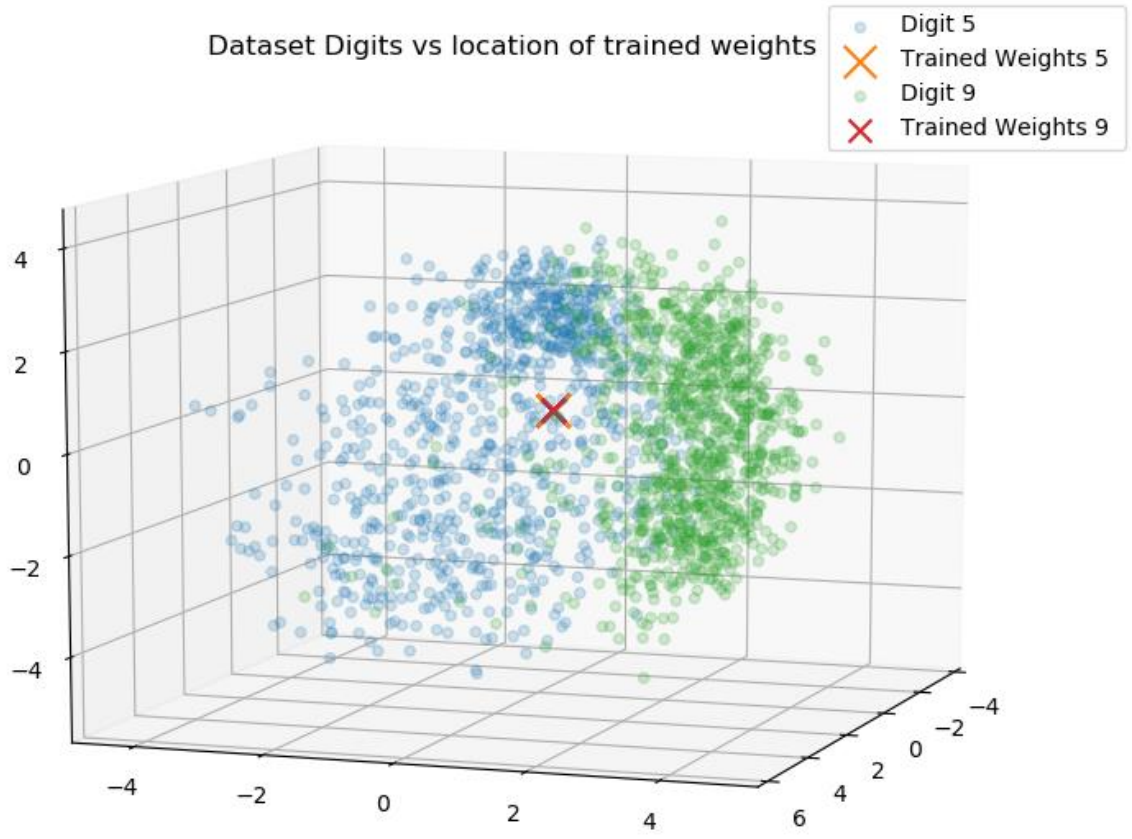


Figure 10 'Stacking' of initial Weights

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 an 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.

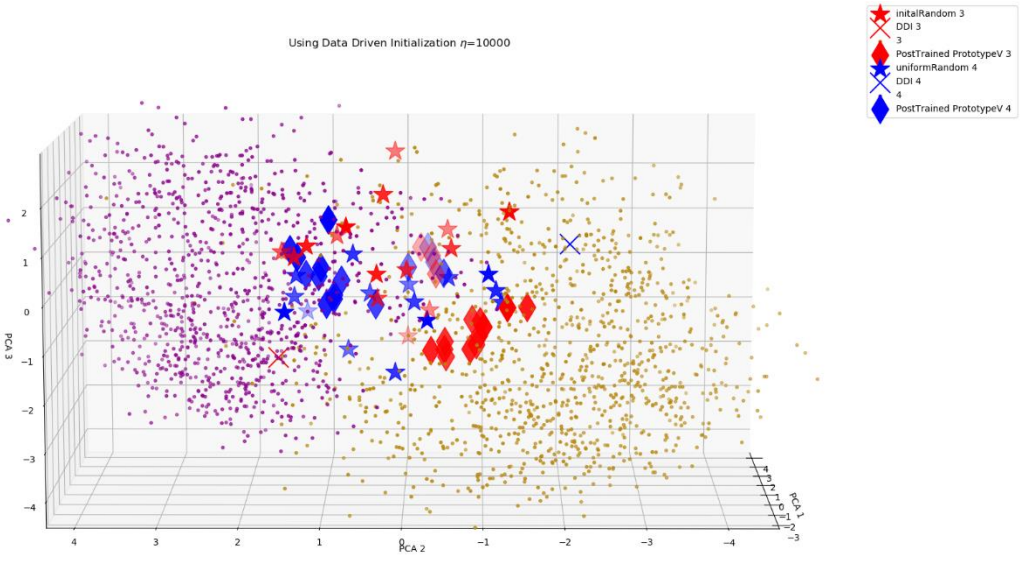


Figure 11 Using Data Driven Initialization

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:

$$\frac{Neurons\ in\ first\ layer}{number\ of\ classes} = neuron\ groups\ in\ second\ layer$$

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 an 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 indexes 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 an 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.

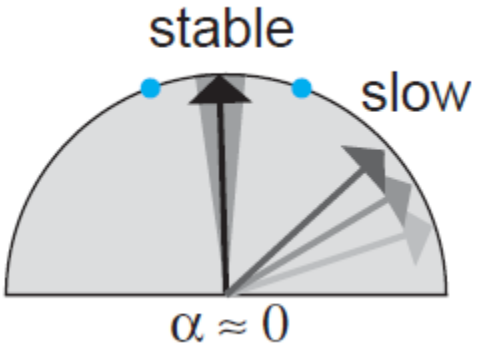


Figure 12 [1] a low learning rate

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.

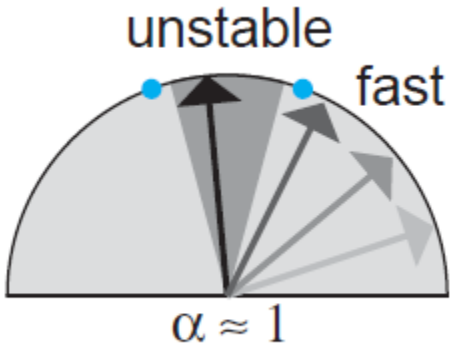


Figure 13 [1] Fast Learning rate

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

At first I approached this problem with trying to find a mean balance of an equilibrium of a learning rate that would fit a medium of low and high learning rate to grab both stable and fast learning. The results

seen below in figure 14 show promise as the network slowly converges

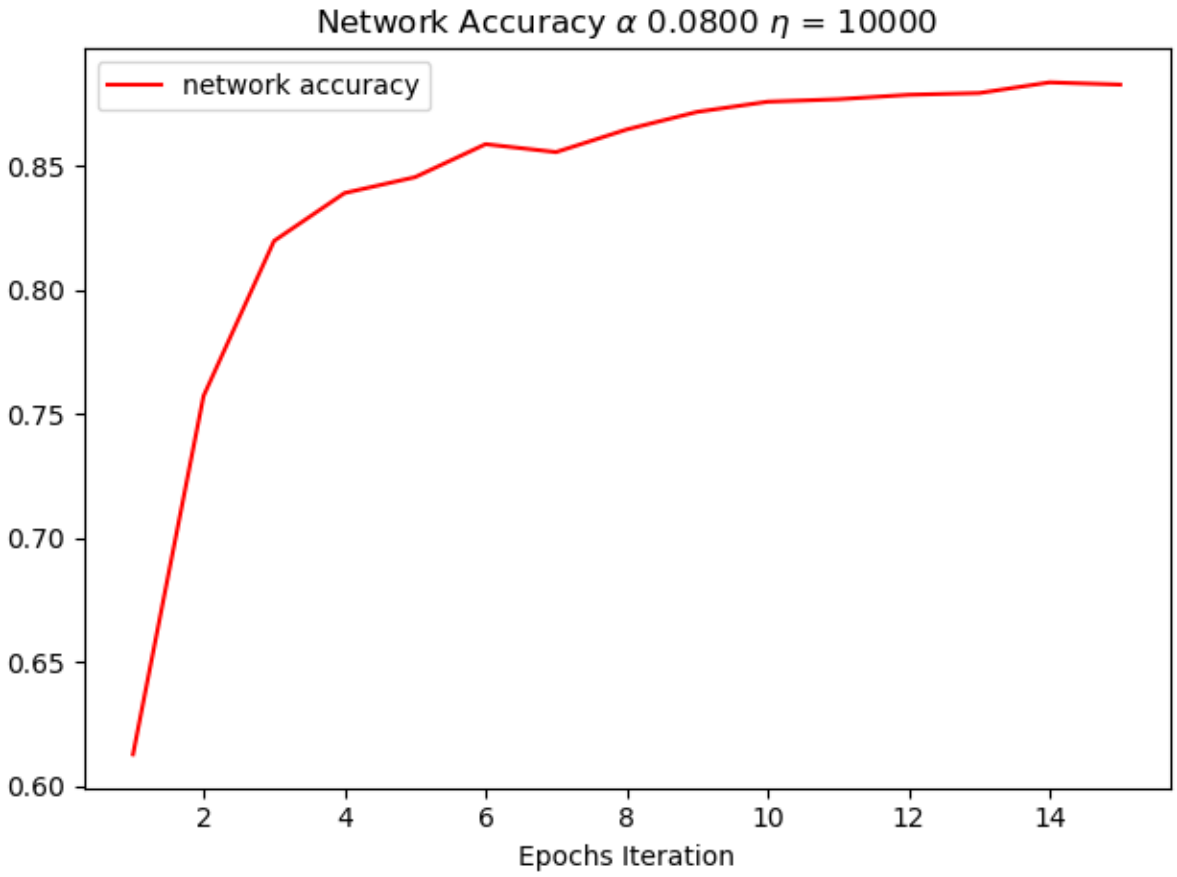


Figure 14 Stable static Learning rate

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

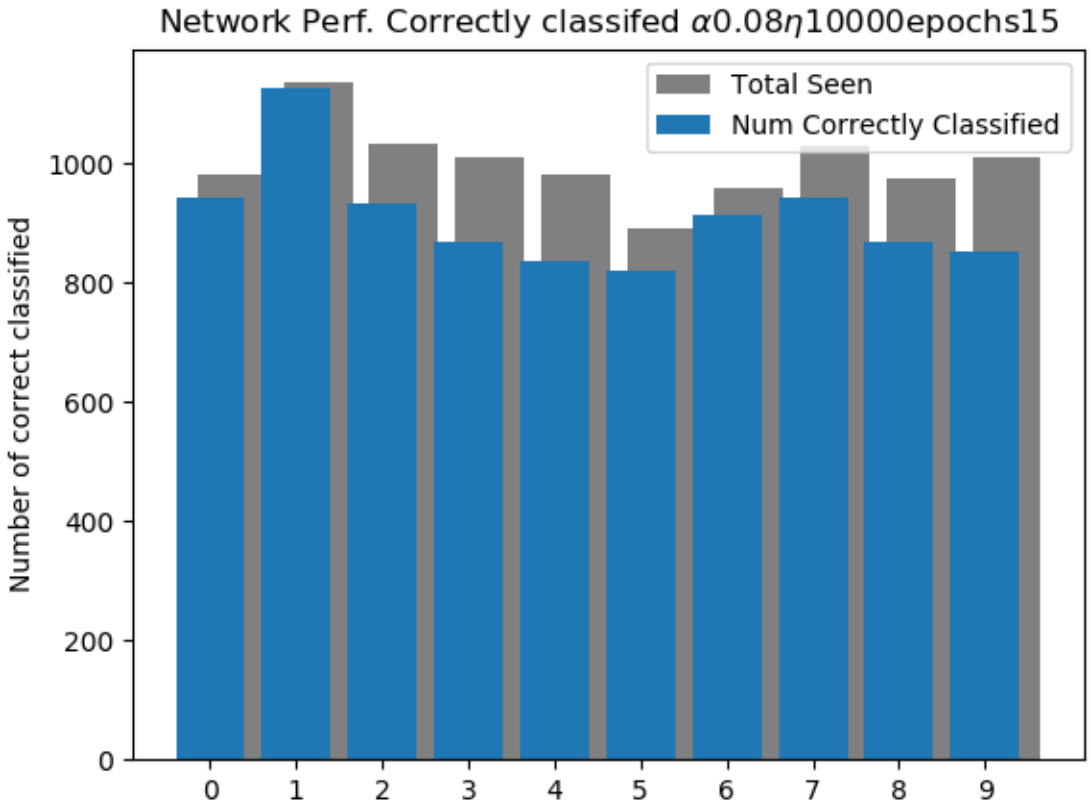


Figure 15 Performance on test set

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 converted 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 a **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 learning 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.

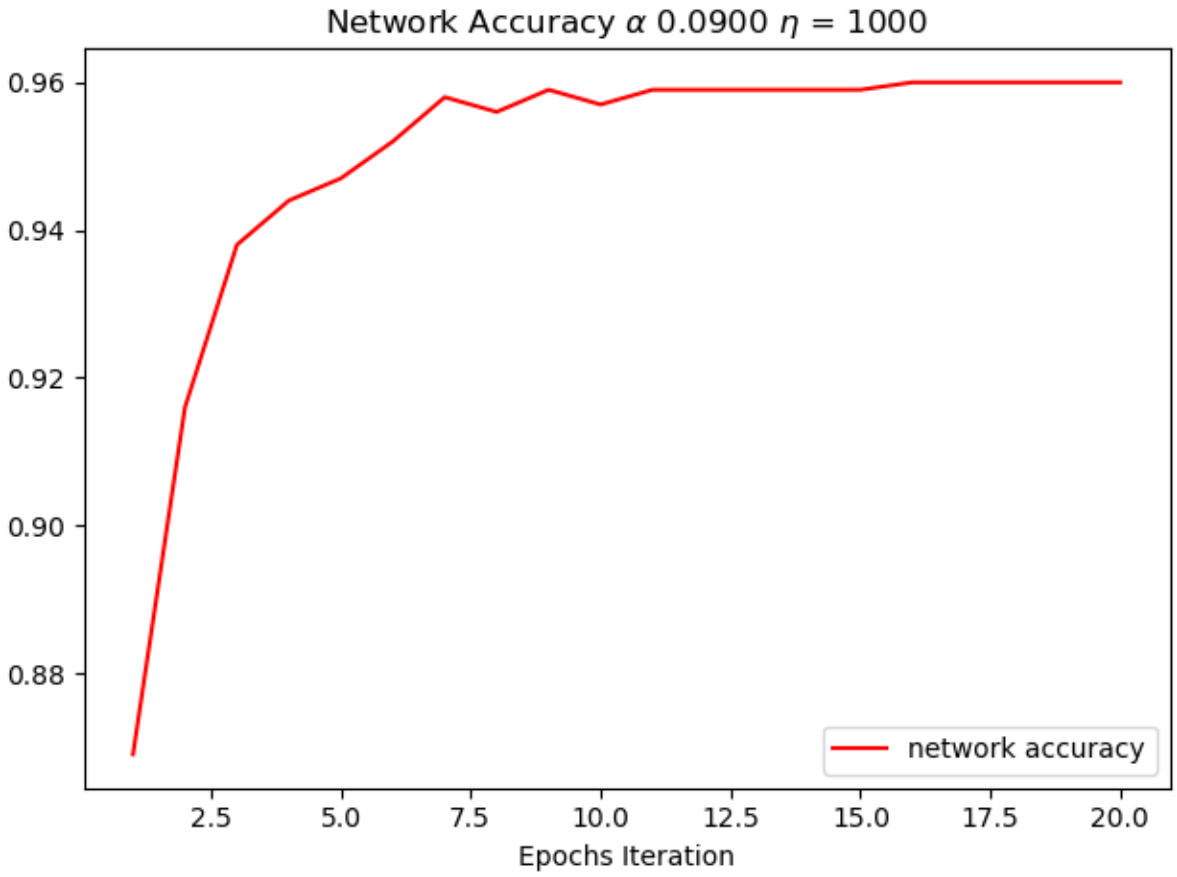


Figure 16 Dynamic Learning Rate

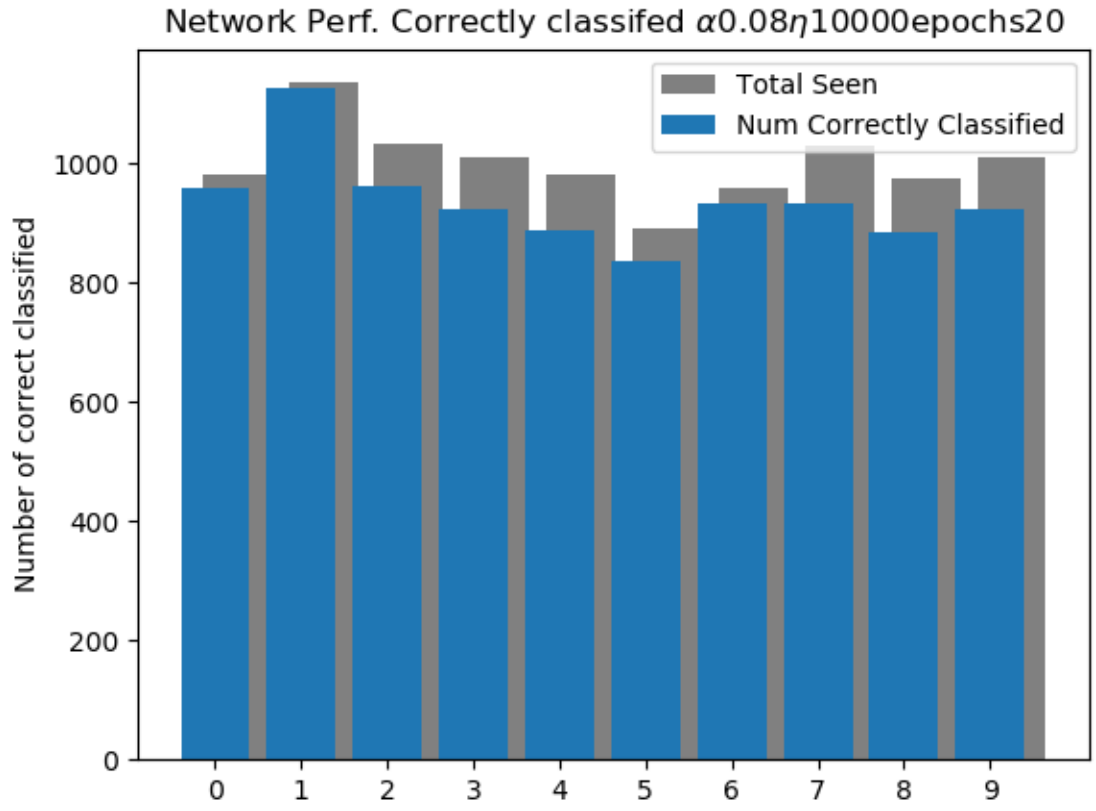


Figure 17 Testing Set Dynamic Learning Rate

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.

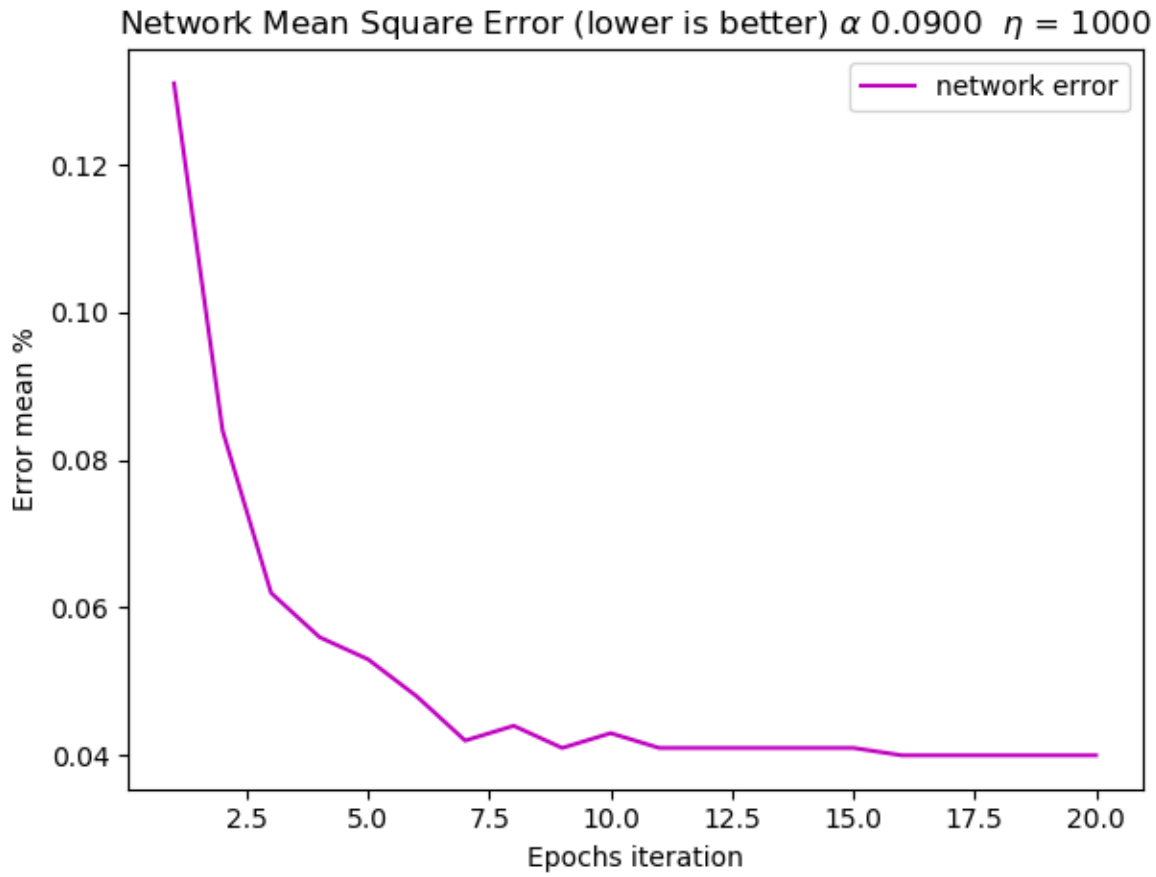


Figure 18 Error of Dynamic Learning Rate

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

During this project, I saw how competitive networks real advantage of being able to use non normalized input vectors, although this feature would be dangerous for some large datasets, it allows the data to retain its original magnitude and scale.

I also learned about the way the competitive network is able to learn suing a dynamic vs static learning rate algorithm, which allows the network to capture groups of clusters better, using generalization. Aside from the performance I also learned about improves that could be done on this network such as improving the learning rule to implementing the learning vector quantization 2 learning rule, this allows 2 neurons to update when a false positive classification is made and a single winner-take all neuron to update if there is a correct classification.

In this network architecture I found 3D graphs where we perform Principle Component analysis the most useful since this network essentially moves neurons closer to their attractive clusters it helps to confirm what the network is actually doing by plotting the neurons position pre-trained vs post-trained.

Conclusion:

During this project I learned that a competitive neural network performs best on data that are non-linear, in example just as the MNIST data set, the distribution was uniform, and there were no(few) linear relationship between each input vector. This is because this competitive network tries to build relationships to the center of the mass of clustered data. If there were relations between each dataset the clusters would cause the performance to suffer because neurons would have a hard time crossing through thin regions of other neurons classification zone.

I also learned for this learning algorithm it is best to use a data driven initialization approach to avoid randomly selecting prototype weights that are nowhere near the mass of input vectors to be learned from.

Extras:

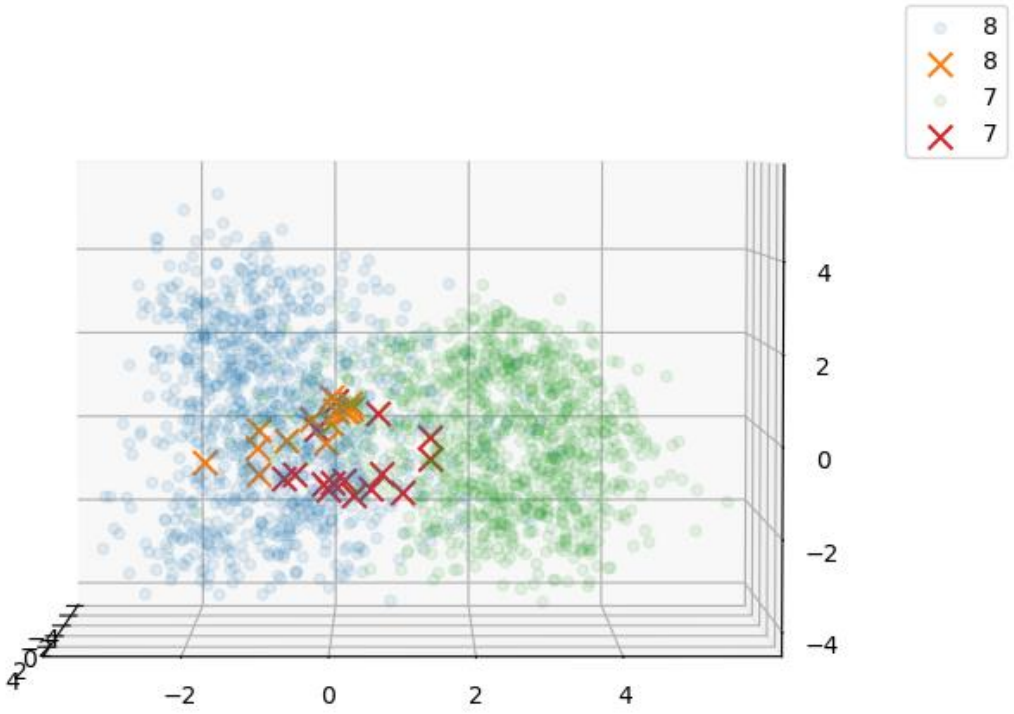
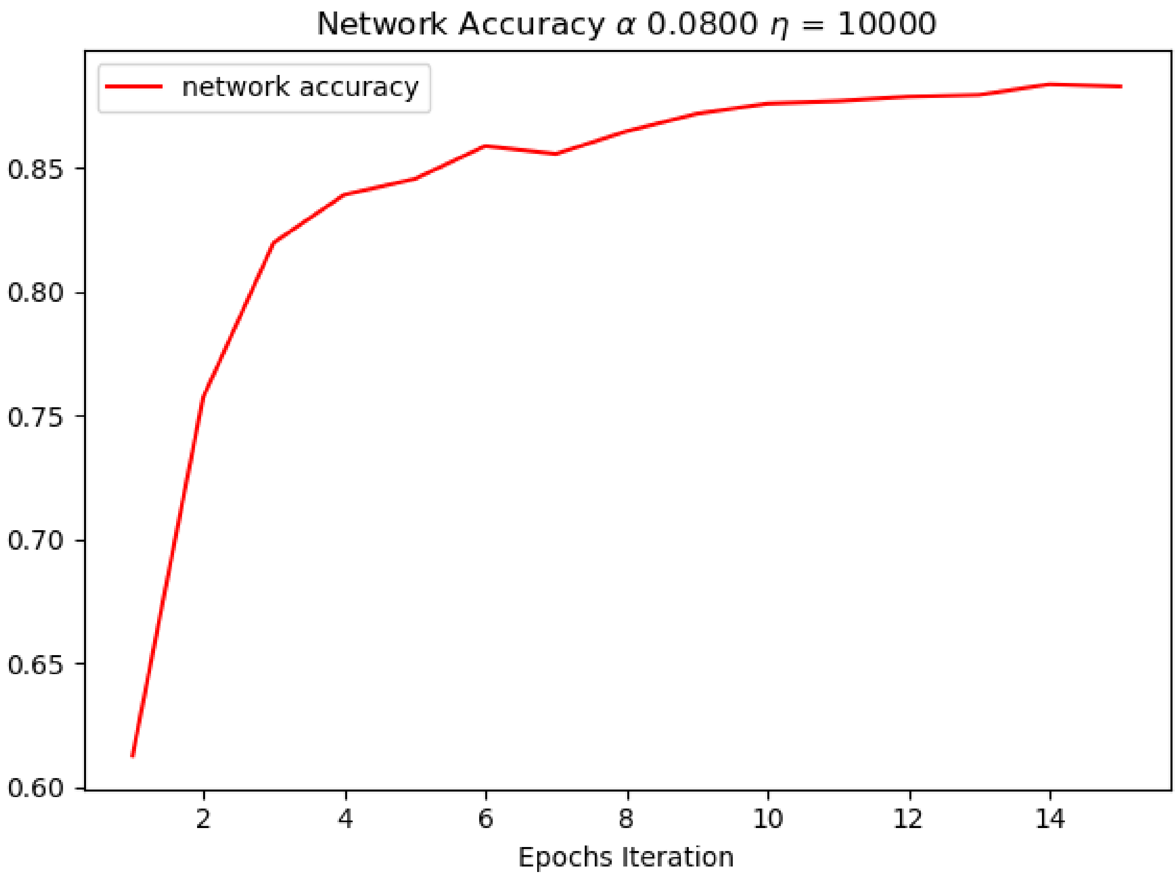


Figure 19 Post Trained Prototype Vectors

In figure 19 we confirm that the competitive network is doing what it should, as we can see the clustering of neurons belonging to digit 8 and 9 group towards their correct digit cluster.

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 epoch that hypothesis is more training with a dynamic decaying learning rate will give us better results.

The learning rate is very sensitive



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 an error rate lets step up the learning rate

Appendix:

Implementation of code

```
#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=
#=#=#\|
#=#| Author: Danny Ly MugenKlaus RedKlouds
#=#| File:   CompetitiveNetwork.py
#=#| Date:   12/6/2017
#=#|
#=#| Program Desc:
#=#|
#=#| Usage:
#=#|
#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=
#=#=#\|

from mnist import MNIST
import numpy as np
import pickle, os, random
from matplotlib import pyplot as plt
from neurolab.trans import  PureLin
```

```
np.random.seed(5)


class Network:
    def __init__(self, epoch=5, learning_rate=.1, verbose=False,
dataDrivenInitalization=True, conscience=False):
        """

        This is a competitive Artificial Neural Network
        using the Learning Vector Quantization method
        Discovered by Kohonen
        Default constructor
        """

        #Testing is 50 subclasses, and 10 final classes
        # with input as a 1 X 784 input vector
        self.classes = 10
        self.subclasses = 300
        self.inputVectorSize = 784
        self.useDDI = dataDrivenInitalization
        self.conscience = conscience

        self.layers = []  # each index is a layer
        self.learning_rate = learning_rate  # the learning rate of the network
        self.epoch = epoch  # number of training iterations

        self._setUpNetwork()

        self.error_results = list()

        self.epoch_error = []
        self.epoch_accuracy = []

        self.makeFinalClassLayer(self.classes, self.subclasses)

    def _dataDrivenInitalization(self, subclasses, classes, imageVector, labelVector):

        assert subclasses % classes == 0
        neuronsPerClass = int(subclasses / classes)
        # each neuron in the group should be a random input vector of the input

        data_X = np.matrix(imageVector)
        data_Y = np.array(labelVector)

        #make a default zero matrix to manipulate
        d = np.matrix(np.zeros((subclasses, self.inputVectorSize)))

        # for the rows 0 -> 4 should be 0 digit 0
        digit = 0
        for i in range(0, subclasses, neuronsPerClass):
            ve = data_X[data_Y == digit]
            for j in range(i, i + neuronsPerClass):
                ridx = random.randint(0, len(ve)-1)
                d[j, :] = ve[ridx]
            digit += 1
        print(d.shape)
        print(d.T.shape)
        assert d.shape == (self.subclasses,self.inputVectorSize)



        self.deadDetection = d.copy()
```

```
        ff = open("detectDead.p",'wb')
        pickle.dump(self.deadDetection,ff)
        ff.close()


        self.layers[0]['weights'] = d.T

        print("finished Data Driven Initialization")
    def _getHotEncoding(self, targetInteger):
        """
        One hot encoding returns a one got encoded value to the
        respective target Integer
        :param targetInteger:  ex 5 will return a 1x10
        ie: 5 -> [0,0,0,0,0,1,0,0,0,0,0]
        :return:  returns a  numpy matrix
        """
        _hot_encoding = np.matrix([
            [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
        return _hot_encoding[targetInteger]

    def makeFinalClassLayer(self, numClasses, numSubClasses):
        """
        For the final output layer we need a layer to combine all subclasses from the
pervious layers to our
        final layer, therefore this method will create a one-hot encoding final layer
for the number of classes/categories
        of the problem we are trying to solve.
        :return:
        """
        assert numSubClasses % numClasses == 0, "Error there is an unbalanced subclass
count please recheck your settings"
        # make the one hot encoded matrix
        hot_encode = np.zeros(shape=(numClasses, numClasses))

        for i in range(len(hot_encode)):
            hot_encode[i][i] = 1
        # we know that we have 0 - 9 digits
        # this hot_encode hold for each index a unique value
        # each index is also associated with each digit starting from 0 -> 9

        subclassPerClass = numSubClasses / numClasses  # get the number of sub classes
per class in final layer

        # assign groups of subclasses to classes
        hot = 0
        m = np.matrix(np.zeros(shape=(numClasses, numSubClasses)))  # make a default
subclass


        for i in range(numSubClasses):
            # m.append(hot_encode[hot])
            if not i == 0 and i % subclassPerClass == 0:
                hot += 1
```

```
            m[:, i] = np.matrix(hot_encode[hot]).T

        assert m.shape == (self.classes, self.subclasses)
        return m

    def _competitiveTrans(self, data):
        r = np.zeros_like(data)
        max = np.argmax(data)  # get the index with the maximum value
        r[max] = 1.0
        return r  # returns the maximum value the winning neurons index
    def _prelinTrans(self, data):
        p = PureLin()
        return p(data)
    def _setUpNetwork(self):
        """
        Our input is a 28 X 28 single row vector, so 784 rows
        Set up a the networks layers,
        generate random weights and bias for the given number of neurons
        :return:
        """

        # so initially we want 50 subclasses so 50 neurons amongst  784 features

        # s is 50 R is 784
        #np.random.seed(0)
        W_1_0 = np.matrix(np.random.uniform(low=0, high=1, size=(
        self.subclasses, self.inputVectorSize)))  # 50 X 784#the input areas
normalized between 0 and 255 or 0,1

        #self.deadDetection = W_1_0.copy()
        print( W_1_0.shape)
        print( W_1_0.T.shape)

        assert W_1_0.shape == (
        self.subclasses, self.inputVectorSize), "Error initializing weights "  #
assert check that these parameters or checkpoints are forced.


        B_1_0 = np.matrix(np.random.uniform(0,.5, (1,self.subclasses)))
        assert B_1_0.shape == (1,self.subclasses)
        layer1 = {'weights': W_1_0.T,  # per following the equations Transpose such
that each colum is a nueron
                  'transFunc': 'Compet',
                  'bias': B_1_0.T}


        W_2_0 = self.makeFinalClassLayer(self.classes, self.subclasses)

        assert W_2_0.shape == (self.classes, self.subclasses)

        layer2 = {'weights': W_2_0,
                  'transFunc': 'Purlin',
                  'bias': 0}

        # populate the layers
        self.layers.append(layer1)
        self.layers.append(layer2)

    def train(self, inputVectors, targetVectors):
        """
        pulbic function to train the network
        :param inputVectors: Given a input matrix
        :parem targetVectors: Given a inputVectors
```

```
        :return: None, Network has been trained
        """

        self.numSamples = len(inputVectors)
        if self.useDDI:

self._dataDrivenInitalization(self.subclasses,self.classes,inputVectors,targetVectors)


        decay_rate = 10
        for iter in range(self.epoch):  # for each epoch
            c_e = 0                     # initialize the current error for this epoch
            for pInput in range(len(inputVectors)):

                # Normalize the input vector by 255.0 ( normalize the input )
                #_P = np.matrix( np.divide(inputVectors[pInput], 255.0) )

                _P = np.matrix(inputVectors[pInput])

               # _P = (_P - _P.min())/ (_P.max() - _P.min())

                assert _P.shape == (1, self.inputVectorSize), "Input vector dimensions
error"


                # get the target integer of the respective input
                targetIndex = targetVectors[pInput]
                # get the one-hot-encoded vector of the respective target input
integer
                _T = self._getHotEncoding(targetIndex).T

                # right now we exact Weights to be 50X784
                assert self.layers[0]['weights'].shape == (self.inputVectorSize,
self.subclasses), "Weights are correct dimensions"

                # calculate the net input of the first competitive layer
                # using Euclidean distance of two vectors
                # -|| dist ||
                netInput_1 = -abs(np.matrix(np.linalg.norm(self.layers[0]['weights'].T
- _P ,
                                                           axis=1)))



                netInput_1 = netInput_1.T  # transpose the results

                ## conscience
                if self.conscience:
                    netInput_1 = netInput_1 + self.layers[0]['bias']
                ############
                # calculate the output of the first layer
                a_1 = self._competitiveTrans(netInput_1)

                ##################### second layer ######################



                assert self.layers[1]['weights'].shape == (self.classes,
self.subclasses), "Error with layer 2 weights"

                assert a_1.shape == (self.subclasses, 1), "First layer shape failed"

                # net input of the second purline layer w^2 * a^1
                # this combines all subclasses to the respective classes in the output
```

```
layer
                netInput_2 = np.dot(self.layers[1]['weights'], a_1)

                assert netInput_2.shape == (self.classes, 1), "net input 2 failed
shape"

                # calculate the output of the second layer,
                a_2 = self._prelinTrans(netInput_2)  # stand them up #10X 1

                # tells us which class this input belongs too

                assert _T.shape == (self.classes, 1), "Error with Target vector"

                # calculate the error

                mse = (np.square(_T - a_2)).mean()



                ########### update weights ###########

                # get the potentially winning neuron

                winning_neuron = np.argmax(a_1)

                # deep copy the net input to modified
                Copy_N = netInput_1.copy()
                # now set the first winning to -inf
                Copy_N[winning_neuron] = -np.inf


                runner_up = np.argmax(Copy_N)  # Get the second winner up from the
copied net input

                correct = np.array_equal(_T, a_2)
                assert self.layers[0]['weights'].shape == (self.inputVectorSize,
self.subclasses), "ERROR with updaing weights"

                #if mse == 0.0:
                if correct:
                    c_e += 1
                    # correctly classified
                    self._updateWeights(_P, winning_neuron)
                else:
                    # false positive, update the runner up
                    self._updateWeights(_P, winning_neuron, runner_up)



            if iter > 1:
                #we have at least 2 iterations
                # check if the error is within a specific distance
                improv = self.epoch_error[iter - 1] - round((1 - (c_e /
len(inputVectors))), 7)
                print("Current improvment %s" % improv)
                if improv > 0.06:
                    self.learning_rate = round(self.learning_rate - .006, 4)
                    print("decreasing LR %s" % self.learning_rate)
                elif improv < 0.0006:
                    self.learning_rate = round( self.learning_rate + .002, 4)
                    print("Increasing LR %s " % self.learning_rate)

            self.epoch_error.append(round(1 - (c_e / len(inputVectors)), 7))
```

```
            self.epoch_accuracy.append(round((c_e / len(inputVectors)),7))


            print("ERROR FOR epoch: %s | Current Error Rate: %s | Numbers Correctly
Classified: %s | Total Patterns seen %s" % (
            iter, self.epoch_error[iter], c_e, len(inputVectors)))
            ################################### WEIGHT UPDATE
#####################################

    def _updateWeights(self, inputVector, winningNeuron, runnerup=None):
        """
        If you give me a runner up that means that there is a WRONG classification
        :param winningNeuron: Int of the winning or false winning neron
        :param runnerup: the runner up if we have a false positive
        :return: None, Weights 2 neurons will be updated if false positive, else
single neuron is updated.
        """
        moveCloser = winningNeuron  # default
        if runnerup is not None:
            # we had a wrong classification move this neuron closer , which is the
runner up neron
            moveCloser = runnerup
            # move the false positive neuron away
            self.layers[0]['weights'][:, winningNeuron] = self.layers[0]['weights'][:,
winningNeuron] - (
            self.learning_rate * (inputVector.T - self.layers[0]['weights'][:,
winningNeuron]))

        # move Closer adjust if we have a runner up then move closer i the runner up
if we do not we default the moveCloser as winning neuron
        # move the runner up closer and the winner farter, we always move something
closer
        self.layers[0]['weights'][:, moveCloser] = self.layers[0]['weights'][:,
moveCloser] + (
        self.learning_rate * (inputVector.T - self.layers[0]['weights'][:,
moveCloser]))

        ######################
        # conscience
        # winning neuron gets its bias reduced while all others are multiplied by a
scalar

        #save the bias for the old bias
        if self.conscience:
            hold = self.layers[0]['bias'][moveCloser]
            fill = np.full((self.subclasses,1),.9)
            #multiply all bias by this scaler
            self.layers[0]['bias'] = np.multiply(self.layers[0]['bias'],fill)
            # penalize the winning neuron
            self.layers[0]['bias'][moveCloser] = hold - .2

    def findDeadNeuron(self):
        #weights currently are col are neurons
        count = 0
        for i in range(self.subclasses):
            if np.array_equal(self.deadDetection.T[:,i],
self.layers[0]['weights'][:,i]):
                count +=1
        return count
    def plotPerformance(self):

        ff = plt.figure(8)
        print("Error: %s" % self.epoch_error)
```

```
        print("Accuracy: %s " % self.epoch_accuracy)
        print("Number of dead Nurons %s" % self.findDeadNeuron())
        x = np.arange(1, self.epoch + 1)



        plt.title(r"Network Mean Square Error (lower is better) $\alpha$ %.4f  $\eta$
= %s" % (
        self.learning_rate, self.numSamples))
        plt.plot(x, self.epoch_error, color='m', label="network error")
        plt.xlabel("Epochs iteration")
        plt.ylabel("Error mean %")
        plt.tight_layout()
        plt.legend()
        fff = plt.figure(23)
        plt.title(r"Network Accuracy $\alpha$ %.4f $\eta$ = %s" % (self.learning_rate,
self.numSamples))
        plt.xlabel("Epochs Iteration")
        plt.plot(x, self.epoch_accuracy, color='r', label="network accuracy")
        plt.legend()

        plt.tight_layout()
        plt.show()

    def predict(self, inputVector, targetVector):
        """
        Test the network's performance to perdict what the given handwritten vector is
        :param inputVector:
        :param targetVector:
        :return:
        """
        _t = np.matrix(self._getHotEncoding(targetVector)).T  # returns the target
        _p = np.matrix(inputVector)
        assert _p.shape == (1, self.inputVectorSize)
        assert self.layers[0]['weights'].shape == (self.inputVectorSize,
self.subclasses)
        assert _t.shape == (self.classes, 1)
        n_1 = -abs(np.matrix((np.linalg.norm(self.layers[0]['weights'].T - _p,
axis=1))))
        a_1 = self._competitiveTrans(n_1.T)
        assert a_1.shape == (self.subclasses, 1)

        # second layer
        assert self.layers[1]['weights'].shape == (self.classes, self.subclasses)
        n_2 = np.dot(self.layers[1]['weights'], a_1)
        assert n_2.shape == (self.classes, 1)
        pur = PureLin()
        a_2 = pur(n_2)  # stand them up # 10 X 1
        assert a_2.shape == (self.classes, 1)

        return np.array_equal(_t, a_2)
    def savePrototypeWeights(self):
        """
        Analytics save the current trained prototypes to a file to analyze
        :return:
        """

        import pickle
        f = open("NetworkWeights.pick",'wb')
        pickle.dump(self.layers[0]['weights'].T, f)
        f.close()
```

```
def pickleData():
    # pickle the data
    # faster load times
    mndata = MNIST('./data')

    images, labels = mndata.load_training()  # loads only the training data
    ################################
    # pickle the data
    imageFile = open('./Sources/images.pickle', 'wb')
    pickle.dump(images, imageFile)
    labelsFile = open('./Sources/labels.pickle', 'wb')
    pickle.dump(labels, labelsFile)
    imageFile.close()
    labelsFile.close()

    ################## Testing


    # images, labels = mndata.load_training()# loads only the training data
    images, labels = mndata.load_testing()
    imageFile = open('./Sources/images_Test.pickle', 'wb')
    pickle.dump(images, imageFile)
    labelsFile = open('./Sources/labels_Test.pickle', 'wb')
    pickle.dump(labels, labelsFile)

    imageFile.close()
    labelsFile.close()
    ############################


def loadPickleData():
    imageVFile = open('./Sources/images.pickle', 'rb')
    imageVector = pickle.load(imageVFile)
    imageLFile = open('./Sources/labels.pickle', 'rb')
    labelVector = pickle.load(imageLFile)

    imageVFile.close()
    imageLFile.close()

    imageVFile_Test = open('./Sources/images_Test.pickle', 'rb')
    imageVector_Test = pickle.load(imageVFile_Test)
    imageLFile_Test = open('./Sources/labels_Test.pickle', 'rb')
    labelVector_Test = pickle.load(imageLFile_Test)

    imageVFile_Test.close()
    imageLFile_Test.close()

    return imageVector, labelVector, imageVector_Test, labelVector_Test


def generateImageVector(imageV, labelV):
    fig = plt.figure(1)
    for i in range(1, 10):
        plt.subplot(3, 3, i)  # 3 rows 3 columns, and subplot number starting at 1
        randomImage = random.randint(0, len(imageV))
        pixels = np.array(imageV[randomImage])  # pick a random image vector to
display
        pixels = pixels.reshape((28, 28))  # reshape the single row vector 784 X 1 to
a square matrix
        plt.imshow(pixels, cmap='gray')  # put the image on the subplot
        plt.title(labelV[randomImage])  # show the label of the image, using the image
vector
    plt.tight_layout()  # prevent label overlays
```

```
def loadData():
    # check if the picked files exist, load them if so, else make them
    if os.path.isfile('./Sources/images.pickle') \
            and os.path.isfile('./Sources/labels.pickle'):
        imagesVector, labelsVector, imageV_test, lablV_test = loadPickleData()
        print("file exist")
        return imagesVector, labelsVector, imageV_test, lablV_test
    else:
        pickleData()
        print("does not exist")


def testLVQNetwork(numSamples, learningRate,epochs):
    # make an instances of LVQNetwork
    # take the testing data,
    # run the test using the 'predict
    # function to test the network against what it gives and what wee expect


    imagesVector, labelsVector, imageVector_Test, labelsVector_Test = loadData()  #
load the data

    generateImageVector(imageVector_Test, labelsVector_Test)

    f = open('./saveNetwork.pickle', 'rb')

    network = pickle.load(f)
    f.close()

    results = dict()
    # initalize dict
    for i in range(10):
        # 0,...,9 inclusive
        results[i] = {'testCount': 0,
                      'correctCount': 0,
                      'classifiedErrors': list()}

    for pattern in range(len(imageVector_Test)):
        # for each testing pattern's index
        result = network.predict(imageVector_Test[pattern],
labelsVector_Test[pattern])
        # true if they are classified correctly false otherwise
        if result == True:
            # corectly classified
            results[labelsVector_Test[pattern]]['correctCount'] += 1  # increment
        else:

results[labelsVector_Test[pattern]]['classifiedErrors'].append(labelsVector_Test[

pattern])  # record the patterns that were miss classified maybe get some insight on
what's going on
        results[labelsVector_Test[pattern]]['testCount'] += 1  # increment number of
times we've tested this pattern
        # print(result)
    print("TOTAL TEST: %s" % len(imageVector_Test))
    # print the results in a

    x = np.arange(0, 10)
    y = list()
    y_total = list()
    for i in sorted(results):
```

```
        y.append(results[i]['correctCount'])
        y_total.append(results[i]['testCount'])
    # print(y)
    print(results)

    ff = plt.figure(6969)

    plt.bar(x + .25, y_total, label="Total Seen", color='gray')
    plt.bar(x, y, label="Num Correctly Classified")
    # plt.subplot(1,1,1)

    plt.xticks(x, ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9'))
    plt.ylabel("Number of correct classified")
    # plt.xlabel("Digit")
    plt.title(r"Network Perf. Correctly classifed $\alpha$%.2f$\eta$%sepochs%s" %
(learningRate,numSamples,epochs))
    plt.legend()

    plt.show()







def main():
    TRAINING = False

    #ep = 15
    ep = 20
    lr = .08
    #lr = 7
    ns = 10000
    if TRAINING:



        imagesVector, labelsVector, imageVector_Test, labelsVector_Test = loadData()
# load the data
        net = Network(epoch = ep, learning_rate = lr, dataDrivenInitalization = True)
        # small learning rate .006 made 9 impossible to find
        num_samples = ns
        net.train(imagesVector[0:num_samples], labelsVector[0:num_samples])


        # save the state of the network for testing
        f = open('./saveNetwork.pickle', 'wb')
        pickle.dump(net, f)
        f.close()

        net.savePrototypeWeights()


        net.findDeadNeuron()
        net.predict(imagesVector[3000], labelsVector[3000])
        net.plotPerformance()
    else:

        testLVQNetwork(ns,lr,ep)
```

```
if __name__ == "__main__":
    main()
```

Acknowledgment

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://stackoverflow.com/questions/17682216/scatter-plot-and-color-mapping-in-python : color mapping in matplotlib

http://www.apnorton.com/blog/2016/12/19/Visualizing-Multidimensional-Data-in-Python/ : data visualizations in python

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. Edition Martin TR. Hagen