Competitve Learning

I. Introduction

This paper discusses the optimal approaches taken to train a one-layer neural network to cluster the set of hand-written characters from the EMNIST dataset. The algorithm implemented to do this uses standard competitive learning to capture as many of the 10 units as possible. In order to do this a method for identifying dead units amongst prototypes was identified and performance measurements were taken over a range of different initial conditions to compare the effectiveness of different techniques.

The competitive learning algorithm that has been implemented works by first loading the contents of the EMNIST dataset. Then a weights matrix is generated, where the rows represent the output neurons and the columns the input neurons. This matrix along with the training data are then normalised (Appendix Fig 1). The algorithm then begins by generating a random index in the input range and choosing a training instance using this random index. Then the index of the neuron in the output firing vector is obtained (k). The change in weights for this k-th output neuron are then calculated using the learning rate in order to get closer to the input. Finally, the weights of the k-th index are updated. This process is repeated over the number of times specified (tmax).

The algorithm that was created makes use of online learning. This is because the weight changes made at a given stage of the algorithm depend specifically on the current training instance that is being taken. This is different to batch learning in which the entire training set is used to compute the weights of the output in one instance (Duchi & Singer, 2009).

In competitive learning it is important to normalize the weight vectors and data at the beginning of the process. This is because the weights from the input will grow without bound and one unit is able to dominate the competition amongst all of the inputs (Goodhill & Barrow). This can be corrected by providing a normalisation step to the inputs so that the weights from the training vectors move directly towards the input vectors, hence forming clusters and deducing patterns.

II. Techniques Implemented

In order to find an optimal solution to capture as many of the different patterns as possible, the following techniques were employed:

A. Noise addition on the weights

This technique applies noise to the weight's matrix after the firing of each winning neuron. The noise applied can be positive or negative noise. In order to do this a noise matrix is created that has the same dimensions as the weight's matrix. A threshold is declared to set the range that the values in noise matrix can take. For competitive learning this noise must be very small in order to encourage other neurons to fire but not change their patterns. Then this noise matrix is added onto the existing weights matrix (Appendix Fig 2).

B. Decaying Learning rate

This technique simply decreases the learning rate within the algorithm as training occurs. In the implemented solution the learning rate decreases after each individual training instance by a very small amount (Appendix Fig 3). This technique has the advantage that the weights are updated by a smaller amount as time progresses to make them converge more closely to the input units.

C. Leaky Learning

This technique involves updating the weights of the losers as well as the winners but with a much smaller learning rate. This ensures that a unit that has always been losing will gradually move towards the input direction until it succeeds in winning a competition (Appendix Fig 4).

D. Updating the winners and neighbouring losers

In this technique the weights of the neighbouring neurons to the neuron that fired are also updated This technique is used to ensure that one neuron does not dominate the competition for firing. In the implementation the neighbours are defined as the decremented and incremented rows of the firing index. However, if the neuron that fires is in the 0^{th} index position then its neighbours are defined as the maximum index position and the 0^{th} index incremented by one. The opposite is true if the firing index is in the maximum indexed position (Appendix Fig 5).

In addition to the above tuning techniques, a method for identification of dead units had to be created. Typically, dead units can be defined as units where the weights either do not change at all or change only very slightly. In this project, a different method was used to identify dead units as neither of these approaches could feasibly be used since all of the neurons fire at least a few times. The implemented method calculates the number of redundant prototypes by looking at the correlations between them. Prototypes that are highly correlated mean that the algorithm has failed to learn differing patterns. As such if a prototype exceeds a certain correlation threshold when compared to any of the other prototypes it is considered to be a redundant prototype. The threshold in the algorithm that determines this is if a prototype is more than 80% similar to one of the previous prototypes that is output that have not been regarded as a redundant prototype (Appendix figure 6). Furthermore, a threshold was set to eliminate prototypes where the sum of the weights was above a particular value. This allowed prototypes such as the one displayed in figure 1 to be classified as a redundant prototype. This was because these prototypes would fire several times but would not learn any useful pattern and so this method allowed such examples to be eliminated. In conclusion, these two methods provide an effective way to identify the prototypes influenced by dead units without the need for visualisation.

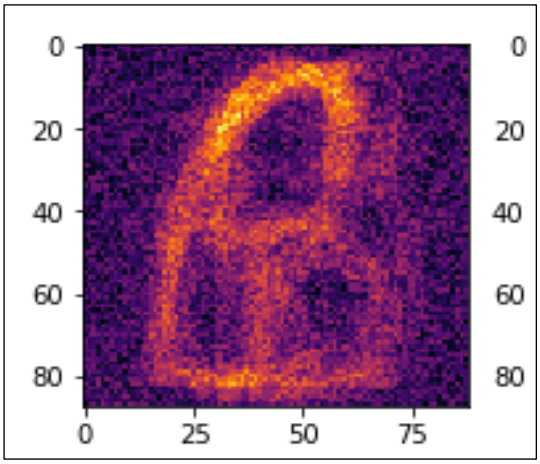

Figure 1: Example prototype of a redundant prototype

III. Performing Measurements

In order to measure the effectiveness of the different techniques they were all assessed based on the average number of redundant prototypes over the same set of initial conditions. As a base line the standard algorithm without any optimization or tuning was also assessed in order to compare the changes that the different techniques made. The results of the different approaches can be seen in the table below (figure 2).

From looking at the results below, it is clear to see that applying the decaying learning rate and updating of neighboring losers techniques resulted in a higher number of average redundant prototypes than the standard algorithm. This suggests that these methods are not effective when it comes to training the network as the prototypes have not learnt sufficient patterns. In the case of the updating of neighboring losers technique a visualization of the prototypes clearly shows that the prototypes are learning from each other as multiple prototypes seem to depict the same letters.

In contrast, the addition of noise and the application of leaky learning each improved upon the average number of dead units when individually applied to the dataset. As a result, further experiments were performed using these techniques combined with other techniques to see if this number would improve even further. As can be seen below, this is not the case and therefore these results suggest that adding noise or applying leaky learning or good tuning method for this dataset.

The full results of these different optimization techniques can be seen in the appendices.

Optimization Technique	**Average redundant ptototypes**
Standard algorithm	5.6
Leaky Learning	5.2
Decaying Learning rate	9.4
adding noise	4.8
update neighbouring losers	13.6
noise and decaying learning	9.7
noise and leaky learning	4.9
decaying learning and leaky learning	9.1

Figure 2: Table of mean number of redundant prototypes

IV. Weight Change as a function of time

In order to track the weight change as a function of time a Graph was created (figure 3) that plots the mean weight change against the time

during the training of the prototypes (appendix figure 6). This is done suing a log-log axes. Since an online version of the algorithm was implemented, it was decided that a moving average the best way to show the weight changes through time. As can be seen from figure 1 the network seems to have sufficiently learnt from the data after around 2500 training instances.

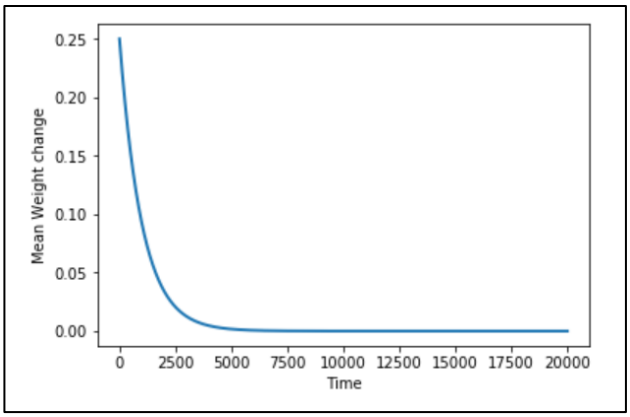

Figure 3: Graph showing the mean weight change as a function of time

V. PROTOTYPES AND CORRELATION

Following the training of the network, the prototypes are displayed visually as can be seen in figure 4. This is done by looping over all of the prototypes (20) and adding them as subplots to the matplotlib figure (Appendix figure 7). The prototypes here represent the patterns that the network has discovered through the competitive learning algorithm relating to the 10 digits from the EMNIST dataset.

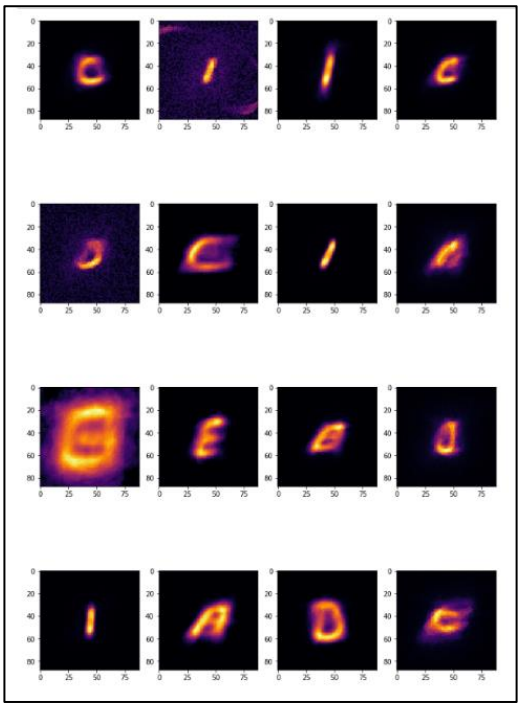

Figure 4: Figure of the visualization of the prototypes

To find the similarities between the prototypes the correlation of the weight's matrix after training was calculated (Appendix figure 8) and consequently plotted as a correlation heatmap to see the relationships between the prototypes more clearly (Figure 5). Here, the darker the cell value, the lower the correlation is and the brighter the cell value the higher the correlation. Typically, very highly correlated prototypes are not good because it means they have failed to learn a range of patterns patterns from the training data.

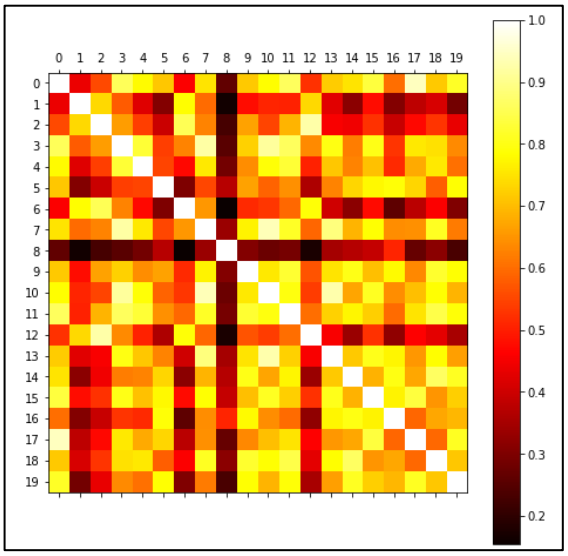

Figure 5: Correlation heatmap of the correlation matrix

VI. Reproducing the results

The results from section three can be reproduced by executing the .py python files contained within the project directory. Each of these files is named in terms of the technique it carries out e.g. the "leaky learning.py" file will produce the results for the leaky learning optimization technique. The results of each technique can be produced in the same manner. Note that these files are used to run the techniques over 10 different initial conditions for the purposes of performance analysis and hence do not include the implemented code for the visualization of the graph, prototypes or correlation matrix. Instead these visualizations can be created via execution of the "Assignment 1.ipynb" file using Jupyter Notebook. This file includes all the optimization approaches in one place along with detailed comments.

VII. CONCLUSIONS

Testing of the different techniques in this report concludes that the best approach to reduce the number of dead units after training the EMNIST dataset with a competitive learning algorithm is to apply noise to the weights after each training instance. This in turn results in the greatest number of letters being clustered. However, it is worth noting that the combination of both the addition of noise and leaky learning improved upon the standard algorithm to an extent similar to that of the noise technique applied solely. Consequently, a further extension to this project could be performing the same techniques over a greater set of initial conditions to see if the combination of the two techniques performs better. In addition, it could also be worth tuning the learning rate of the leaky learning and weights of the noise to see if more optimal results can be found.

References

Duchi, J., & Singer, Y. (2009). Efficient Online and Batch Learning Using Forward Backward Splitting. *Machine Learning Research*, 2899-2934. Retrieved from https://core.ac.uk/download/pdf/82969135.pdf

Goodhill, G. J., & Barrow, H. J. (n.d.). The Role of Weigt Normalization in Competitive Learning. *Neural Computation*, 255-269.

Appendices

```
[n,m]  = np.shape(train)                    # number of pixels and number of training data
#number of pixels is 7744 (88*88)
#number of training data examples is 7000

eta    = 0.05                              # learning rate
winit  = 1                                  # parameter controlling magnitude of initial conditions
alpha = 0.999

tmax   = 20000                             #number of training instances
digits = 20                                #number of digits

wMatrix = winit * np.random.rand(digits,n)        # Weight matrix (rows = output neurons, cols = input neurons)
normW = np.sqrt(np.diag(wMatrix.dot(wMatrix.T)))
normW = normW.reshape(digits,-1)  # reshape normW into a numpy 2d array
wMatrix = wMatrix / normW                          # normalise using numpy broadcasting -  http://docs.scipy.org/doc/numpy-1.1

#make a copy of the inital weights matrix
INITIAL_W=np.array(wMatrix)

counter = np.zeros((1,digits))              # counter for the winner neurons
wCount = np.ones((1,tmax+1)) * 0.25         # running avg of the weight change over time

#normalise the dataset
norm = np.atleast_1d(np.linalg.norm(train, ord =2, axis =0))
norm[norm==0] = 1
train = train / np.expand_dims(norm, axis = 0)
[n,m]  = np.shape(train)

#check train data is normalized
print(numpy.linalg.norm(train[:,1]))
```

Appendix Figure 1: Code snippet of the parameters and variables for the algorithm and the normalization process

```
#adding noise to weights
mu, sigma = 0, 0.00005                              #set a noise value threshold
noise = np.random.normal(mu, sigma, wMatrix.shape) #generate a noise matrix the same size as the weight matrix
wMatrix[k,:] += noise[k,:]                          #apply the noise matrix to the weights matrix
```

Appendix Figure 2: Code snippet of the noise generation and applying noise to weights

```
#decaying learning rate
eta = (eta/tmax)*(tmax-t/1000)          #learning rate decays for each output firing until tmax
```

Appendix Figure 3: Code snippet for the implementation of a decaying learning rate

```
### leaky learning
for s in range(0, digits):                      #loop over the number of neurons
    if (s != k):
        low_eta = eta / 1000                   #set a learning rate that is much lower than that of the winner neuron
        dw = low_eta * (x.T - wMatrix[s,:])     #get the change in weight for neuron index s
        wMatrix[s,:] += dw                      #update the weights of the s indexed neuron
```

Appendix Figure 4: Code snippet for the implementation of leaky learning

```
#update neighboring losers
if (k-1 >= 0):                          #get the index of the next to the winning neuron
    dw = eta * (x.T - wMatrix[k-1,:])   #get the weight change for the k-1 neuron
    wMatrix[k-1,:] += dw                #update the k-1 neuron
else:
    dw = eta * (x.T - wMatrix[digits-1,:])
    wMatrix[digits-1,:] += dw
if (k+1 < digits):
    dw = eta * (x.T - wMatrix[k+1,:])
    wMatrix[k+1,:] += dw
else:
    dw = eta * (x.T - wMatrix[0,:])
    wMatrix[0,:] += dw
```

Appendix Figure 5: Code snippet for the implementation of updating neighboring losers

```
[i,j]  = np.shape(correlation_matrix)
counter = 0
deadUnits = np.zeros((1,digits))

#sum all the weights for each prototype
sums = wMatrix.sum(axis=1)

#if the sum of a prototypes weights is greater than 40 it is considered a dead unit
for y in range(j):
    if (sums[y] > 40):
            deadUnits[0,y] += 1
            counter +=1

#calculates the number of dead units by recursing through the correlation matrix and
#comparing the correlation between prototypes. If a prototype is has a threshold higher than
#0.8 when compared to a previous non-dead unit is classified as a dead unit.

for y in range(j):
    if (deadUnits[0,y] == 0) :
        for x in range(i):
            if (deadUnits[0,x] == 0):
                if (correlation_matrix[x,y] > 0.8 and correlation_matrix[x,y] != 1):
                    deadUnits[0,x] += 1
```

Appendix Figure 6: Code snippet for determination of a dead unit

```
# Plot running average
plt.plot(wCount[0,0:tmax], linewidth=2.0, label='rate')
plt.ylabel('Mean Weight change')
plt.xlabel('Time')
plt.show()
```

Appendix Figure 7: Code snippet for the plotting of weight change as a function of time

```
#plot all the prototypes
w=15
h=15
fig=plt.figure(figsize=(12, 24))
columns = 4
rows = 5
for i in range(1, columns*rows+1):
    fig.add_subplot(rows, columns, i)
    plt.imshow(wMatrix[i-1,:].reshape((88,88), order = 'F'),interpolation = 'nearest', cmap='inferno')
plt.show()

# Plot a prototype
#plt.imshow(W[10,:].reshape((88,88), order = 'F'),interpolation = 'nearest', cmap='inferno')
```

Appendix Figure 8: Code snippet for plotting a visual for all the prototypes

```
#calculate correlation matrix
correlation_matrix = np.corrcoef(wMatrix)

#plot the correlation matrix
plt.figure(figsize=(8,8))
plt.matshow(correlation_matrix, fignum=1, cmap='hot')
plt.xticks(range(len(correlation_matrix[0])))
plt.yticks(range(len(correlation_matrix[0])))

# Add colorbar
plt.colorbar()
plt.show()
```

Appendix Figure 9: Code snippet for the calculation of the correlation matrix and its plotting as a heatmap

	Seed Value										
Optimization Technique	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Average**
Standard algorithm	8	6	5	7	9	2	5	6	4	4	5.6
Leaky Learning	5	4	6	7	4	6	2	5	8	5	5.2
Decaying Learning rate	7	9	11	6	11	11	10	10	9	10	9.4
adding noise	5	3	6	3	6	6	4	4	5	6	4.8
update neighbouring losers	13	14	14	12	12	15	12	16	13	15	13.6
noise and decaying learning	11	10	12	9	9	11	9	7	11	8	9.7
noise and leaky learning	2	7	4	4	7	3	4	10	1	7	4.9
decaying learning and leaky learning	10	9	6	9	9	10	9	10	8	11	9.1

Appendix Figure 10: Full results for the different optimization techniques, showing number of redundant prototypes for each seed and average number of redundant prototypes for each technique