Competitive Learning

Department of Computer Science
University of Sheffield

***Abstract*—This report describes how competitive learning is used to cluster the data from the EMNST dataset, which is not normalised. The aim of this report is getting the fewest dead units as possible by implementing different techniques by itself or as combinations. From the experiments, adding the noise showed the best performance with the fewest dead units shown from the prototypes compare to other techniques and with 20 clusters showed the good balance of the number of prototypes and optimising the neural network.**

***Index Terms*—Implementation, Optimisation, Consideration, Conclusion**

I. Introduction

Competitive learning is one form of machine learning that finds the patterns of data with given input variables only (Unsupervised learning). The aim of competitive learning is finding the patterns of input data to cluster the outputs. This is achieved by competing for each of the output neurons from each input and updates the only one winning neuron. The winning output unit has the highest activation to the given input pattern, therefore, the weights of winners move closer to the input pattern, the surrounding neurons are unchanged. Because of such rule, the strategy of the competitive algorithm is also called as winner-take-all. Figure 1 shows briefly how the network is created. The circles the left represents the input neurons N i.e the features of input and the circles on the right represents the output neurons M i.e cluster of input.

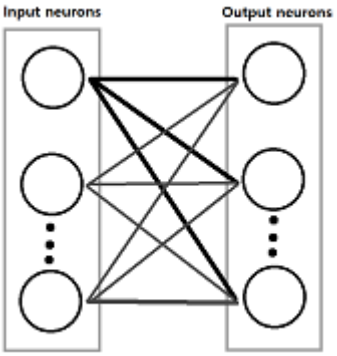

Figure 1. Brief description of how the Neural Network is created with input and output neurons are fully connected.

II. Dataset and algorithm implementation

A. EMNIST dataset

The dataset used in this paper is EMNIST dataset.The EMNIST dataset contained 7000 characters from A to J with 88 x 88 pixel images for each character.

B. Implementation

The main steps for competitive learning are following:

1) Create the weight matrix of output neurons with randomly generated weights.
2) Choose the random pattern of input.
3) Calculate the activation of each output neurons.
4) Update the weight of winning neuron only.
5) Repeat from step 2 until any of following condition is met.
   - Weights do not change.
   - Weight changes less than a set threshold.
   - Maximum number of iterations has been reached.

From the 7744 x 7000 input neuron N, the network is fully connected with 10 (A-J characters) output neurons M, stored in 10 x 7744 matrix. This matrix is called weight matrix W. From this process, it is important to normalised both dataset and weight matrix W, which will be discussed in the next section of this paper.

To update the winning neuron's weight in step 4, the initialized weight matrix W and output of firing neuron from trained data has to be determined from step 1–3. The rows of the weight matrix are called prototype. They are the samples that fire the most from the matrix and by visualizing them, we can find out the algorithm is implemented well or not. The update weight rule for step 4 is the following:

$$\Delta W_k = \_learning\_rate(x - W_K) \tag{1}$$

Where $\_learning\_rate$ is the learning rate, x is the input pattern and W_K is output neuron with the highest firing vector. The input pattern x is determined by picking a training instance using a randomly generated index in the input range. W_K is determined by k-th output neuron of weight vector, which k is the index of the firing neuron calculate by argmax of W_x. W_x is the dot product of the weight vector for the x and the pattern of it.

III. Optimisation

The aim of this paper is to identify the best technique(s) that gives the fewest dead units of prototypes after the training. Dead units are the units that do not move because they never become the winners and therefore there are nothing to learn. The number of counters for the winner unit had fired in the last epoch is used to identify the dead unit. Originally, the only dead units are if the counter number is zero, which means none of the neurons has moved. However, in this paper, we set when

the counter is less than 50 is also dead unit. The following is the techniques that are implemented for experiments:

1) Setting the random values (by Gaussian Distribution) as initial conditions of weight vectors (Baseline).
2) Setting the random samples from the dataset as initial conditions of weight vectors (Selected weight).
3) Adding noise to the output of neurons (Noise).
4) Decaying the learning rate for each iteration (Decaying learning).
5) Updating the weights of the winners and the neighbouring losers (Update).

The conditions of the experiments are following:

- The techniques are implemented both individually and with combination to identify the best and worst technique(s) as single or combined usage.
- The initial learning rate is 0.05.
- The initial iteration is 5000.
- Other than selected weight technique, the way of selecting the weight vectors of each technique are the same as 'baseline'.
- Each configuration were run 10 times. Whenever the implemented techniques are changed, the training completely runs again.
- The experiment repeated for each chosen amount of output neurons, which are 10, 20, 30, 40 and 50.

The results are displayed in Table 1, showing mean and standard deviation of dead units after 10 running for applied techniques in leftmost row of table with cluster of top column of table.

Techniques \ Clusters	10	20	30	40	50
Base	Mean: 0.9 STD: 1.29	Mean: 8.2 STD: 1.40	Mean: 18.4 STD: 0.97	Mean: 28.4 STD: 1.50	Mean: 38.5 STD: 1.18
Noise	Mean: 2.5 STD: 0.85	Mean: 6.0 STD: 1.15	Mean: 13.5 STD: 1.27	Mean: 22.0 STD: 1.49	Mean: 31.0 STD: 2.21
Decaying	Mean: 0 STD: 0	Mean: 9.7 STD: 1.06	Mean: 20.3 STD: 1.34	Mean: 30.7 STD: 0.95	Mean: 40.7 STD: 1.16
Update	Mean: 0.2 STD: 0.42	Mean: 7.8 STD: 1.14	Mean: 18.9 STD: 1.37	Mean: 28.9 STD: 1.10	Mean: 38.7 STD: 1.25
Selected weight	Mean: 0 STD: 0	Mean: 0.6 STD: 0.84	Mean: 2.3 STD: 1.34	Mean: 5.6 STD: 2.83	Mean: 10.4 STD: 2.99
Noise + Decaying	Mean: 0.1 STD: 0.32	Mean: 5.0 STD: 1.63	Mean: 14.7 STD: 1.16	Mean: 23.4 STD: 1.17	Mean: 33.8 STD: 1.69
Decaying + Update	Mean: 0.2 STD: 0.42	Mean: 10.0 STD: 1.29	Mean: 20.6 STD: 0.52	Mean: 30.6 STD: 12.6	Mean: 41.2 STD: 1.23
Noise + Update	Mean: 0.1 STD: 0.32	Mean: 5.2 STD: 1.48	Mean: 14.9 STD: 1.79	Mean: 23.5 STD: 1.65	Mean: 32.6 STD: 1.65
Noise + Decaying + Update	Mean: 0 STD: 0	Mean: 5.2 STD: 2.04	Mean: 14.9 STD: 1.10	Mean: 24.0 STD: 1.56	Mean: 32.0 STD: 1.89

Table I
TABLE OF RESULTS WITH APPLYING DIFFERENT TECHNIQUES ON COMPETITIVE ALGORITHM

The result shows the best technique when implemented as single is selected weight technique. This is because while weight vectors of other techniques are random values, the selected weight's weight vector is the actual sample from the dataset. However, this can lead prototypes captured with the same information as the weight vector is specified. The decayed learning rate showed the worst result as single implemented technique and updating the neighbouring losers was the second worst. Excluding the result when it is 10 clusters, the means of dead units of each cluster were more than baseline. The only technique that worked well other than the selected weight was adding noise, except when the cluster is 10. Updating technique did well when the clusters are lower number, but showed the worse result as soon the cluster numbers get higher than 20.

Most of the combinations of techniques worked better than baseline. The best combination was noise, decaying learning and update with showing a better result than baseline in all clusters. The combination of decaying learning and the update was the worst since most of the results of it were worse than the baseline. This combination is not only the worst as combination of techniques, but also shows the worst result within the table. The other two combinations of techniques had similar results, but noise and decaying learning rate combination had a lower standard deviation. From the result, we could see that any techniques that combined with noise showed good result. The only combination without the noise showed the worst result from the experiments.

The amount of dead units in both 10 and 20 clusters are within 50%. As soon as the number of clusters goes up to 30, half of the prototypes were shown as dead units. For 50 clusters, it even showed 60 to 80% were dead units, which strongly proves it's too big number. All the results when cluster is 10 were good, however, it's not suitable number to show variety of data. Therefore, the right amount of clusters to use in this paper is 20.

To sum up, optimizing the dead units by adding noise only with 20 clusters showed the best to do both optimizing the fewest dead units and prototyping.

IV. CONSIDERATION

A. Batch rule? Online rule?

Both batch rule and online rule is about which type of learning rule the algorithm is. The batch rule means when the algorithm updates the weight for each epoch, keeps the weight while computation. On the other hand, the weights for online rule constantly updated for each sample while computation. Therefore, the algorithm in this paper is the online rule.

B. Normalisation

The purpose of normalisation of the dataset is to avoid the duplicate data by sorting them. Within the raw dataset, there

might be same data repeating, without the normalisation, if the algorithm is proceeded then it will lead the result biased. Also by normalisation, all the data can be compared fairly as different character has different amount of data.

For the similar reason, the purpose of normalisation of weight vector is to make them comparable fairly. The mathematical equation of getting the firing rate is the dot product of array, which equals to $\|a\|\|b\|cos(\theta)$. Each $\|a\|$ and $\|b\|$ is weight vector and the training instance using the randomly generated index. As the training instances are the same for all output neurons, this means the size of weight vector is the only factor that makes neuron more likely to fire, regardless how much angle there is between them. Any neurons with small weight vectors will never win even though they have better angles between them and this will lead to producing more dead units, which is against the aim of this paper. By normalisation of weight vector, all the weight vectors get equal size i.e $\|W\| = 1$, hence all the neurons will be compared fairly. Therefore, normalisation for both dataset and weight vectors are critical to achieving the aim of this paper.

C. Average weight change over time

The weight change over time is shown in Figure 2 in log-log scale. It is clear to see from the graph that the weight changes drop significantly from 10^4 and around 50^4, the changes become very small. This means the network already learnt enough so that the prototypes are moved to its closest samples, the further epochs would make no difference to the result.

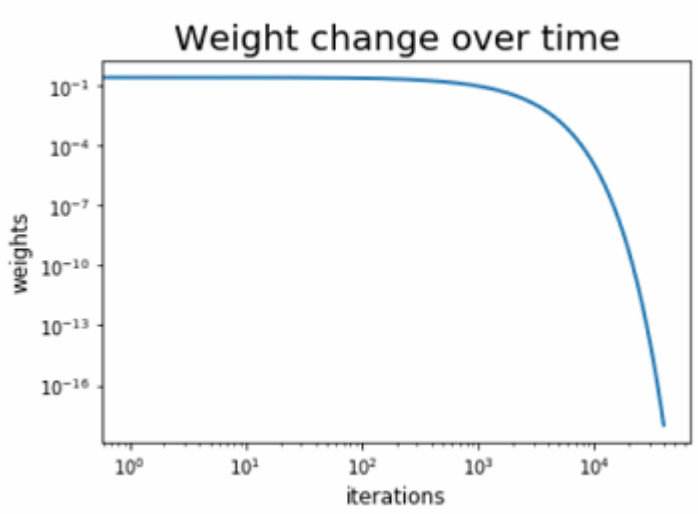

Figure 2. The average weight change over time. As the iteration goes, the smaller the weight changes are, able to expect the more training will lead even smaller weights change.

D. Prototypes

Figure 3 and 4 is the example of 30 prototypes with ineffective techniques applied. Figure 3 is part of Figure 4 with a closer look. Figure 5 and 6 is the part of the example of 50 prototypes with effective techniques applied. Figure 5 is part of Figure 6 with a closer look. The numbers above the prototypes of all Figures imply the counter of the neurons moved for that prototype. With comparing both figures, prototypes in Figure 5 is more clearly presented than those of Figure 3, with higher counter numbers on average. Some prototypes represented the same character but in a different way. For instance, the 11th and 12th prototype in Figure 5 is both represents English character 'G' but in a different style of writing. 9th, 13th and 14th prototype of Figure 5 represent English character 'C' not only the style of writing but also the size is different. With comparing Figure 4 and 6, the more prototypes displayed, the more variety of clusters can be checked.

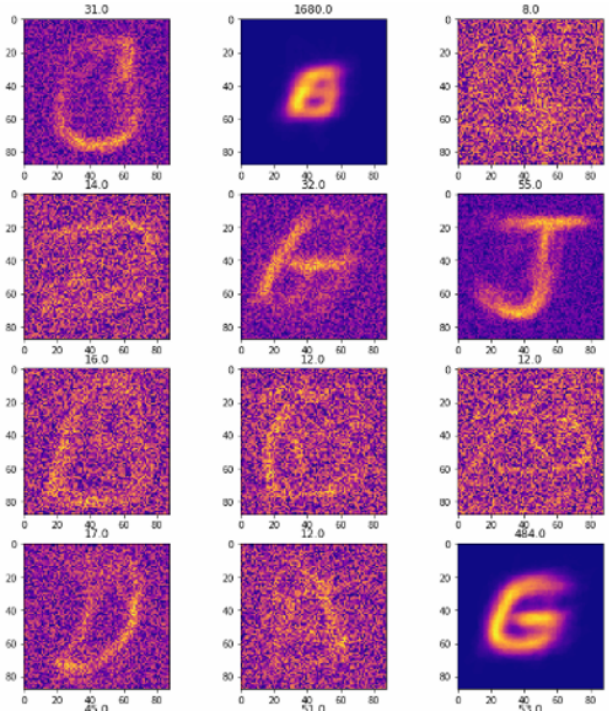

Figure 3. The part of example of 30 prototypes with badly implemented techniques

E. Correlation Matrix

To see the similarity of each prototypes, the correlation matrix is used. Figure 7 is the 20 prototypes with implementing adding noise technique with its correlation matrix. The mathematical formula of correlation matrix is following:

$$Corr_(i,j) = \sum_{n=o}^{N} z_i^n z_j^n / N \tag{2}$$

where N is number of prototypes, n is sample in dataset and z_k is active state of prototype (either 1 or -1). The formula produces N x N matrix at the end. the active state is decided by whether the prototype fired more than threshold or not. If fired more than threshold, marked as 1, otherwise marked as -1. Figure 7 shows one of the example, with threshold is 0.35. From the Figure 7, visualised 20 prototypes and the correlation

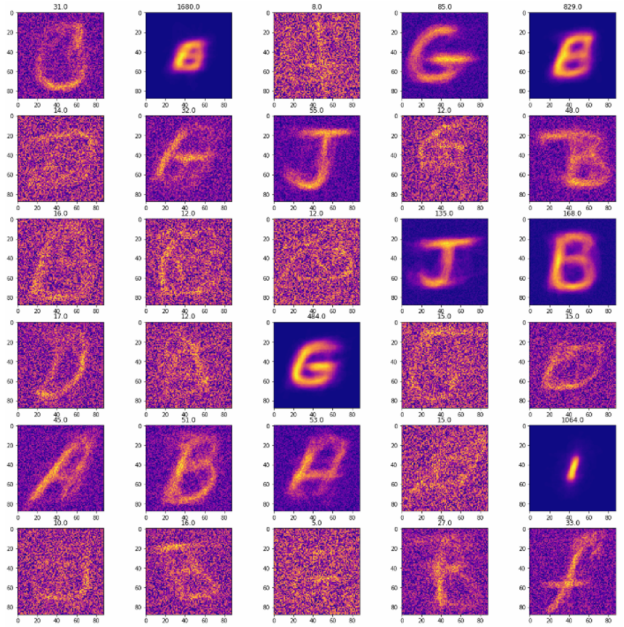

Figure 4. The example of 30 prototypes with badly implemented techniques

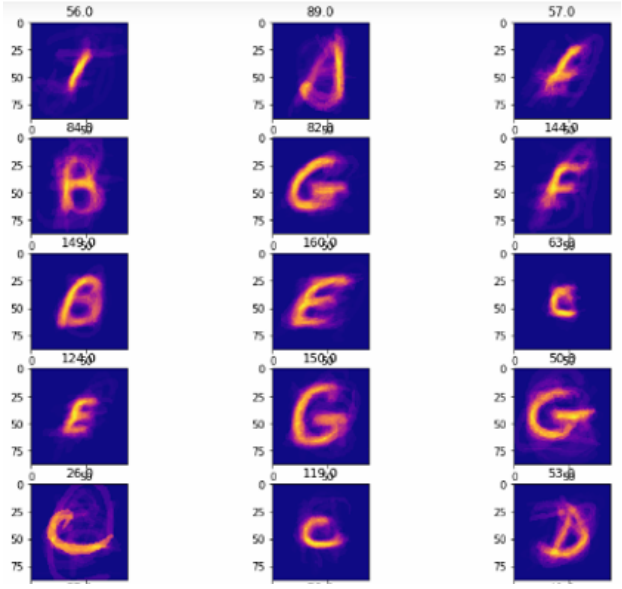

Figure 5. The part of example of 50 prototypes with implementing effective techniques

of each prototypes can be checked, easily compared between them.

V. Conclusion

The aim of this paper is to find out the effectiveness of different techniques implemented on the competitive algorithm and which one is the best for getting the fewest dead units. Some techniques worked well. Especially the weight selection and adding noise. Different cluster numbers also effected on the result, as the more prototypes required, the more classes of characters and dead units are visualized. Competitive learning

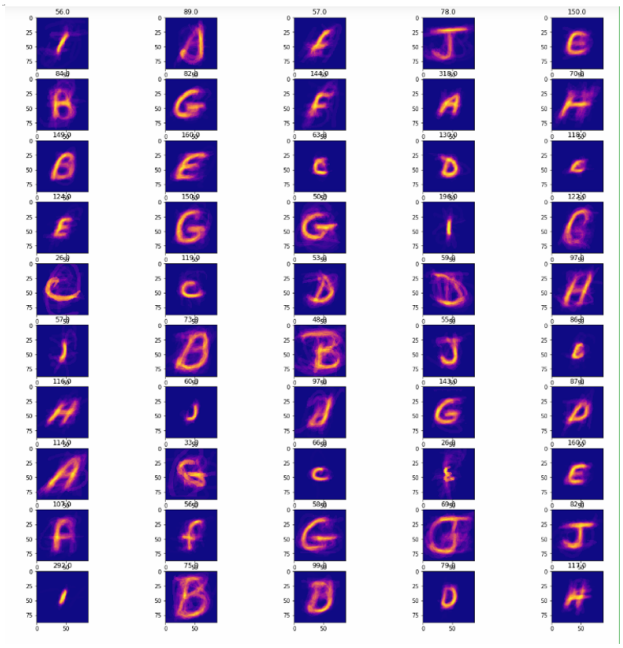

Figure 6. The example of 50 prototypes with implementing effective techniques

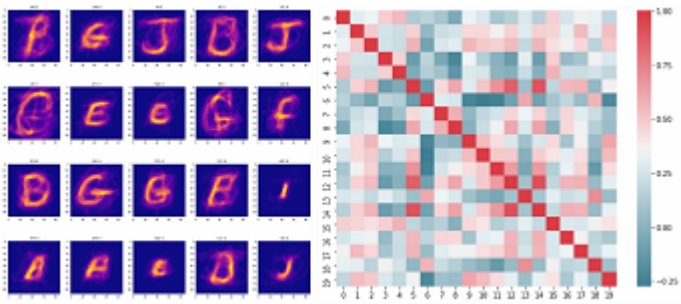

Figure 7. Prototypes and its correlation matrix. The image on the left is the visualized 20 prototype with implementation of adding noise and weight selection and the image on the right is the correlation matrix when the threshold is 0.35

worked well in this paper, however, if the number of iteration changes the whole result could be changed and that also could be an important factor of reducing the number of dead units as the more iterations, the more the algorithm learn.

Appendix A
Reproduce Results

To reproduce the results and figures about visualise prototype, count dead units, weight change over time with implementing different techniques and visualise correlation matrix, run 'Assignment_competitive_learning.ipynb' via 'jupyter notebook'.

Appendix B
Code

Algorithm 1 Normalise data

```
♯ Labels not needed as unsupervised learning approach not used
train = np.genfromtxt (Name_of_file.csv)
♯ Number of pixels (input neurons) and number of training data
[n,m] = np.shape(train)
♯ Normalise the dataset
normT = np.sqrt(np.diag(train.T.dot(train)))
train = train / np.matlib.repmat(normT.T,n,1)
data = train.T
♯ Check the normalisation is correct
♯ If the value is 1.0 or 0.9999, normalisation is correct
numpy.linalg.norm(data[1,:])
```

Figure 8. Data is normalised to reduce the duplicated data. Without normalisation, there likely the result is biased.

Algorithm 2 Weight initialisation

```
♯ Weight initialised randomly (base)
if tech_weight ≠' on' then
    W = np.random.rand(output_neurons, n)
else if tech_weight ==' on' then
    ♯ Weight initialised with selection (tech_weight)
    indices = np.arange(data.shape[0])
    ♯ Choose output neurons amount of random indices
    selected = np.random.choice(indices, output_neurons)
    W = data[selected]
```

Figure 9. Weight initialised either randomly or with selected samples from the dataset depends on whether weight selection technique is activated or not.

REFERENCES

[1] Bernd Fritzke. Some competitive learning methods. page 16, 05 1997.

Algorithm 3 Weight Normalisation

```
♯ Reshape into a 2d array
normW = np.sqrt(np.diag(W.dot(W.T))).reshape(output_neurons,-1)
♯ Normalise weight
W = W / np.matlib.repmat(normW.T,n,1).T
```

Figure 10. Weight vectors are normalised before getting firing output for fair comparison. all the size of weight vectors become 1.

Algorithm 4 Learning and implementation

```
♯ Pick training instance using random index in input range
trained = train[:, math.ceil(m * np.random.rand()) − 1]
♯ Get output firing
similarity = W.dot(trained)/output_neurons
♯ Add noise (tech_noise)
if tech_noise ==' on' : then
    ♯ Generate noise and add it to similarity
    noise = np.random.rand(output_neurons, 1)/200
    similarity = similarity + noise
end if
♯ Get the index of the firing neuron
fire_idx = np.argmax(similarity)
♯ Decaying the learning rate (tech_decay)
if tech_decay ==' on' : then
    ♯ Calculate the dacaying learning rate
    learning_rate = initial_learning_rate * (final_learning_rate/initial_learning_rate) * *(t/tmax)
    ♯ Calculate the change in weight using online rule
    change = learning_rate*(trained−W[fire_idx,:])
else if tech_decay ==' off' : then
    ♯ If decaying learning rate technique not implemented, use fixed normal learning rate
    change = initial_learning_rate * (trained − W[fire_idx,:])
end if
♯ Update the weight
W[fire_idx,:] = (W[fire_idx,:] + change)
♯ Update the surrounding losers weights with much smaller value (tech_update)
if tech_update ==' on' : then
    W = W + (change * 0.0005)
end if=0
```

Figure 11. Implementation of online rule with different techniques. First get the output firing of each output neurons. Find the most similar neuron with the highest similarities and calculate the change in weight. This change will then applied to winning neurons and repeated until there is no change in weight or weight changes less than threshold or iteration is done. Between each process, different kinds of techniques are applied if they are activated. For the decaying learning technique, the exponential decaying learning rate is chosen according to [1]. Ignore =0 at the end of algorithm

Algorithm 5 Weight change

```
♯ Running average of the weight change over time
wCount = np.ones((1,iteration+1)) * 0.25
♯ % weight change over time (running average)
wCount[0,each_iteration_of_loop] = wCount[0,each_iteration_of_loop-1] * (0.999 + change.dot(change)*(1-0.999))
```

Figure 12. To find the weight changes over time, each weight changes are for each iterations.

Algorithm 6 Counting Dead units

```
♯ Counter for the winner neurons
counter = np.zeros((1,output_neurons))
♯ Increment counter for winner neuron
counter[0,fire_idx] += 1
```

Figure 13. Count the movement of winner neurons of prototype. If the number of neurons moved are less than 50, the prototype is distinguished as dead unit.

Algorithm 7 Correlation Matrix

```
corr_matrix = np.zeros((digits,digits))
for i in data do
    output = np.dot(W, i)
    shape = np.zeros_like(output)
    shape[output ≤ threshold] = -1
    shape[output > threshold] = 1
    corr_matrix += np.outer(shape, shape)
end for
corr_matrix /= data.shape[0] =0
```

Figure 14. Calculate the correlation between the neurons. The output of each neurons are calculated and compared with threshold, which is 0.35. Ignore =0 at the end of algorithm