A Competitive Learning Algorithm to Cluster the Set of Hand-Written Characters

The University of Sheffield

Hyun Han (160152230)

hhan5@sheffield.ac.uk

Abstract

It was a dilemma to obtain an accurate classifier in the computer science field, since various machine learning algorithms were introduced. The project aims to build a sophisticated competitive learning algorithm on a one layer network to cluster a set of hand-written characters. Based on the standard competitive learning algorithm provided in advance, several optimisation methods and tuning process will be implemented to capture 10 characters with as few dead units as possible.

INDEX TERMS machine learning, competitive learning, dead units, tuning, and optimization for a network.

1. INTRODUCTION

A reduced dataset called EMINST which contains 7000 characters from A to J, where each character is an 88x88 pixel image. Each column corresponds to elements (the number of characters) and row to the feature vectors. The formal template provided the standard competitive learning algorithm to be implemented further. However, the given training dataset is not normalized, while the weight vector is already normalized. Therefore after the normalization process is handled, some optimisation skills will be covered. After those experiments, the best judgement will be made to identify dead units and the strong evaluations and analysis will follow.

2. BATCH VS ONLINE

An online processing focuses on handling transaction when they occur, so the user can obtain a direct output, where it allows the user to keep a constant interaction with the system [1]. On the other hand, batch processing is ideal to handle a large amount of data, where the input files of the data are collected in a batch and they are processed together. As works in batch processing are divided into sub jobs and then processed, it does not require a huge amount of hardware resources [2].

In this paper, the algorithms utilise the online system, because each time the winner neuron is detected, when given a random input patterns, some tuning process are implemented; the weight of the winner neuron is updated, the weights of all the other loser neurons are updated but with a much smaller learning rate, the weight of neighbour neurons of the winner neuron are updated and lastly the learning rate is decreased every iteration in a certain degree.

2. NORMALISATION

The normalization process, as a pervasive technique in machine learning areas, is essential to provide an appropriate dataset that a classifier can use. The process aims to remove a distortion effects in the range of feature values in the dataset or losing information. For instance, if one of the feature vectors in a dataset is with values ranging from 0 to 1, there can be a potential problem during modelling process, when combining it with other feature vectors with values fluctuating between two numbers larger than 0 and 1. Normalization is the process that tries to maintain the general distribution and ratios across the feature vectors via creating new values [3].

The dot product has been a common measure of the similarities of the input vector to output node weight vector; however several problems are introduced in this method. Firstly, it does not consider a case where some input vectors have different magnitude regardless of their direction information. Secondly, it is necessary that all the weight vectors must have the equal magnitude to calculate the similarity between the input vectors and the weight vectors. Those problems can be resolved by using Euclidian distance measure instead of using dot product as a similarity measure [4].

Given a set of input patters E^u is applied to the algorithm, the winner neuron i among the output neurons fires. Then, the weight Wij for the winning neuron is updated closer to its input vector, so the winning neuron can win next time again, if the same input is given. The formula $\Delta Wij = n(E^u_j - Wij)$ shows the weight update process called standard competitive learning rule. Simply, the gap between the input vector and the winning neuron can be multiplied by a learning rate n. [5]

In this paper, the training dataset provided and weigh vectors are normalized based on the sum of weight vectors components as shown in Figure 1. The norm command was used for the validation of the success of the normalisation. Euclidian method was tried to be applied, but it could not be done because of the time constraint.

3. ADDING NOISE

We know that if we change the data arbitrarily in a program, it can change the entire results. However, if only small amounts get modified, it does not result in a totally different result. Adding noise to inputs by small amounts is a sort of a training process that makes an algorithm not change its output if the input gets changed by a tiny amount. It makes the algorithm more robust to avoid the potential overfitting problem [6].

In this paper, whenever each element of the weight vector is accessed by using nested for loops, a random number between five and six with a random sign is generated and added to its own value in order to add noise, as shown in Figure 2. It was not clear to obtain the best range, but it is turned out that the numbers between five and six lead to the best outcome, after some experiments, although noise is traditionally a number between 0 and 1.

4. DECAYING LEARNING RATE

Learning rate can be said as sensitivity that decides how drastically the weight will be updated. Throughout the modelling process, the algorithm gets closer to the target point. However, if the learning rate does not change, it is likely to go over the target point. It will not reach a global maximum. Therefore, it is important to decay learning rate on a time scale.

As shown in Figure 2, the learning rate divided by the number of iteration is subtracted from the learning rate throughout the modelling process. That is, the learning rate will be zero at the last iteration process.

5. LEAKY LERANING

The standard competitive learning algorithm aims to update only the winner neuron's weight, but it is more realistic to reason that all the other neurons still have to be updated but in a much smaller scale. The figure 3 shows that all the other loser neurons are updated with 0.1% of the learning rate on the current process.

6. UPDATING NEIGBOURS

In this paper, the neighbours of the winning neuron is updated as well, as it is reasonable to think that the closer neurons to the winning neuron will have a bigger weight change than the other loser neurons. In this paper, a judgement was made that the weigh update in those neighbours should be smaller than the winning neuron's change. In the Figure 4, the learning rate is timed by 0.01, so the learning rate will decrease by 99% of its value. If the first or the last neuron among the output neurons is chosen as the winning neuron, only one neighbour next to them is updated.

7. DEAD UNITS

Dead unit means neurons that do not fire at all throughout the modelling process in a classic definition. However, it is turned out that all of the neuros at least fire more than once. Therefore, the intuitive approach was made to decide the standard for identifying dead units. The Figure 5 explains the calculation used in this process. The average firing rate can be calculated by dividing the entire iteration tie by the number of output neurons. As the algorithm is a classic model, 0.5 is multiplied as a decreasing factor. Any neurons that fire less than this standard are regarded as dead units. They are plotted as a histogram to grasp the entire trend of the output neuros.

8. RESULTS

The table 1shows the performance improvement when all methods discussed so far are applied. Ten different initial conditions were tested to understand any exception results. Although it is proved that the standard algorithm works better for the seed 1000, the one with all optimisation methods applied works better overall across all different seed conditions.

9. AVERAGE WEIGHT CHNAGE OVER TIME

The figure 6 shows the average weight changes over time when the standard algorithm runs. It decreases from the beginning until 5000. After then, there is no significant change.

The figure 7 shows the average weight changes over time when all the optimisation methods are applied. Until around 2500 the average changes surge but then after then it decrease. After 5000, it is unlikely to see a further decrease in the figure. The reason of the sudden increase in the beginning part is because of adding noise with large numbers.

10. PROTOTYPES

Visualising the prototypes can be the best way to check the performance of the algorithm. The Figure 8 and 9 show sixteen prototypes that some of them are the same alphabet, since there are only ten alphabets to display from A to J. In figure8, it is hard to see alphabet D and H, while there are four similar B, three J, two E and two G. In theory, the prototypes that have the same alphabet should have high correlation. However, as shown in correlation matrix, it is hard to tell that the prototypes of the same alphabet have a significant high correlation values when compared to the other alphabets. It is because the same alphabets also can have different sizes or shapes. Hence, when given a small shape of A, a small shape of C will more likely to have higher correlation value with the A than a large shape of A.

The figure 9 is the prototypes when all the optimisation techniques are applied. Now as shown in the figure there is only one alphabet that cannot be captured, although the 8th one seems like C. There are some prototypes that cannot be identified as any of alphabets, but still the figure can display more alphabets than when the standard algorithm was used.

11. CONCLUSIONS

The standard learning algorithm displayed quite clear prototypes, but it could not capture two alphabets. When the optimisation methods are applied, some of the prototypes were quite hard to be identified as an alphabet, but one more alphabet was captured. It is true that there were less dead units in the algorithm with the optimisation methods, yet the result can vary depending on how to define the definition of dead units. As it was hard to capture a significant performance increase with the optimisation methods, those methods should be reconsidered, when considering those extra methods take more memory power.

HOW TO REPRODUCE THE RSUTLS

In the third box, there is a line saying "np.random.seed(number)". The initial conditions can be decided by replacing "number" to a number you want as a seed.

In the fourth box, there is code for adding noise to weights. This code can be commented out to test the algorithm without any extra methods.

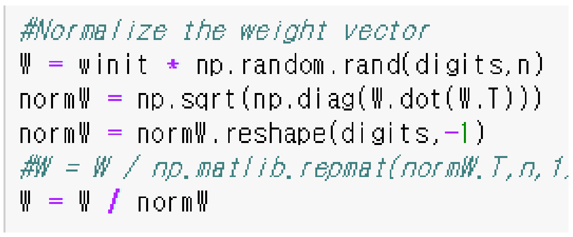
The fourth box contains leaky learning, neighbour updating and decaying learning. Theses as well can be commented out to test the standard algorithm.

ACKNOWLEDGEMENT

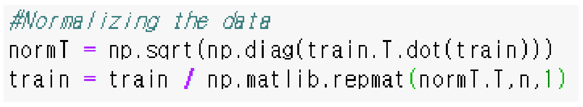
This work was done as a first assignment task for a module called 'Adaptive Intelligence' taught by Eleni Vasilaki.

APPENDIX

Figure1. By using dot product, normalize the training dataset given and the weigh vector

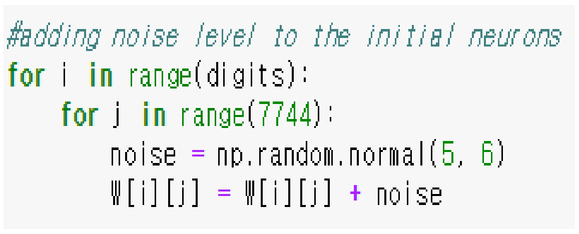


```
#Normalize the weight vector
W = winit * np.random.rand(digits,n)
normW = np.sqrt(np.diag(W.dot(W.T)))
normW = normW.reshape(digits,-1)
#W = W / np.matlib.repmat(normW.T,n,1,
W = W / normW
```



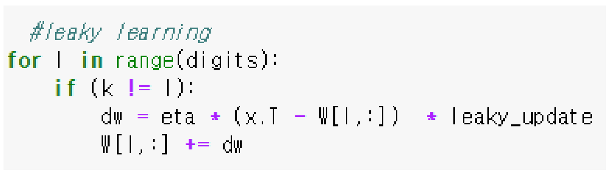
```
#Normalizing the data
normT = np.sqrt(np.diag(train.T.dot(train)))
train = train / np.matlib.repmat(normT.T,n,1)
```

Figure 2. Add random numbers between 5 and 6 with a random sign are added to each element of the weight vector.



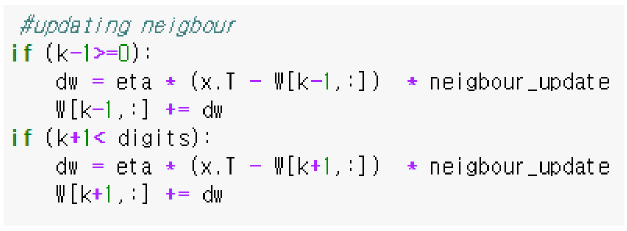
```
#adding noise level to the initial neurons
for i in range(digits):
    for j in range(7744):
        noise = np.random.normal(5, 6)
        W[i][j] = W[i][j] + noise
```

Figure 3. Update all the other lose neurons but with a much smaller scale in terms of the learning rate. The variable ‘leaky_update’ is 0.0001.



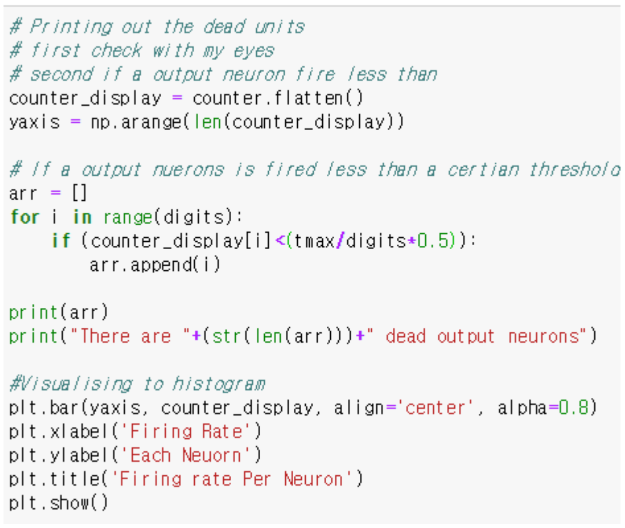
```
 #leaky learning
for l in range(digits):
    if (k != l):
        dw = eta * (x.T - W[l,:])  * leaky_update
        W[l,:] += dw
```

Figure 4. The weights of the neighbours of the winning neuron are updated but with a smaller scale. The variable ‘neigbour_update’ is 0.01



```
 #updating neigbour
if (k-1>=0):
    dw = eta * (x.T - W[k-1,:])  * neigbour_update
    W[k-1,:] += dw
if (k+1< digits):
    dw = eta * (x.T - W[k+1,:])  * neigbour_update
    W[k+1,:] += dw
```

Figure 5. Plotting a histogram of the firing times for each output neuron. If there are neurons that fire less than half of the average firing times, they are regarded as dead untis.



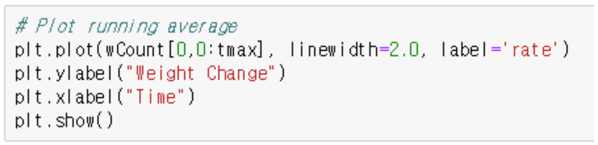
```
# Printing out the dead units
# first check with my eyes
# second if a output neuron fire less than
counter_display = counter.flatten()
yaxis = np.arange(len(counter_display))

# If a output nuerons is fired less than a certian threshold
arr = []
for i in range(digits):
    if (counter_display[i]<(tmax/digits*0.5)):
        arr.append(i)

print(arr)
print("There are "+(str(len(arr)))+" dead output neurons")

#Visualising to histogram
plt.bar(yaxis, counter_display, align='center', alpha=0.8)
plt.xlabel('Firing Rate')
plt.ylabel('Each Neuorn')
plt.title('Firing rate Per Neuron')
plt.show()
```

Figure 6. Average weight changes over time for the standard competitive learning algorithm.



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

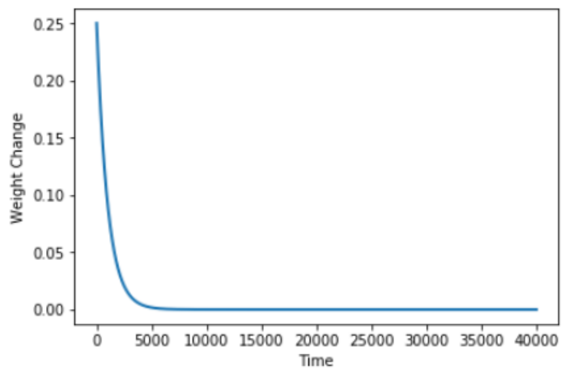
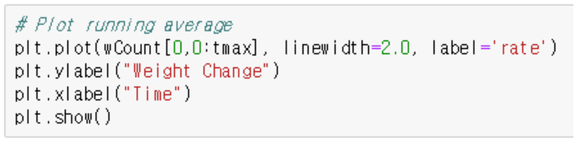


Figure 7. Average weight changes over time for

the algorithm with all the optimisation skills applied.



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

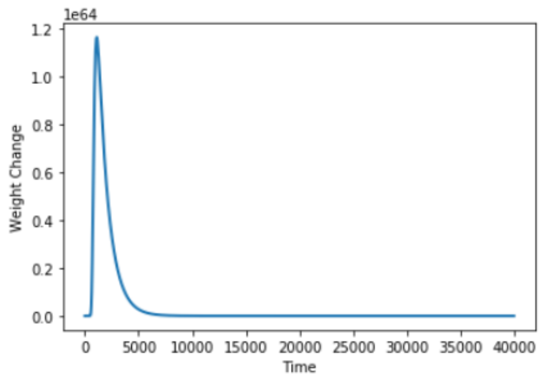
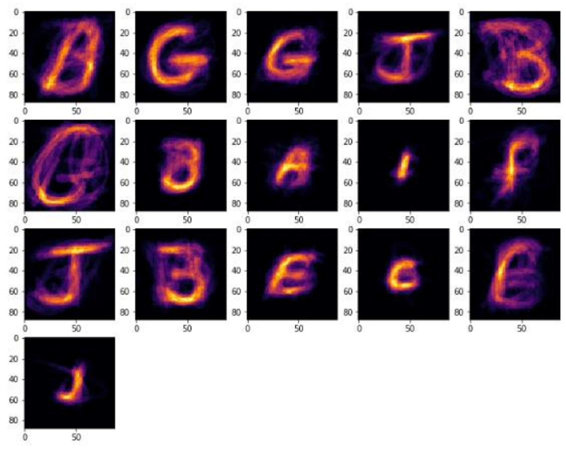


Figure 8. Prototypes and correlation matrix for the standard algorithm with seed 1000



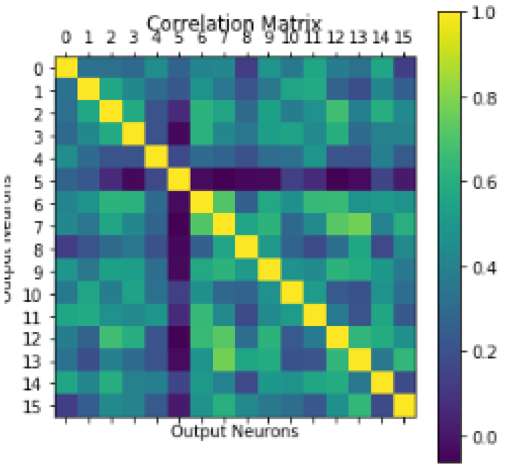
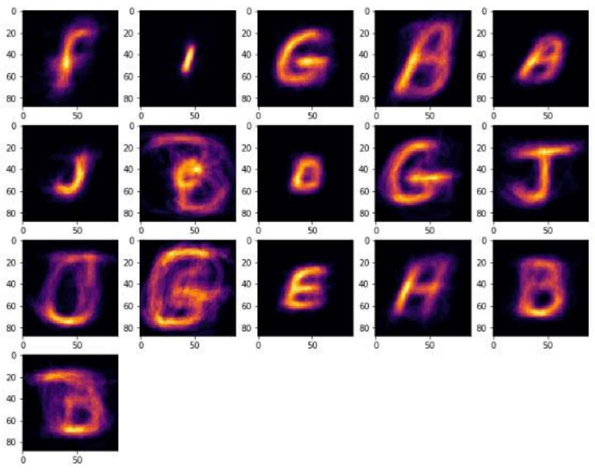


Figure 9. Prototypes and correlation matrix for the algorithm with all the optimisation skills are applied



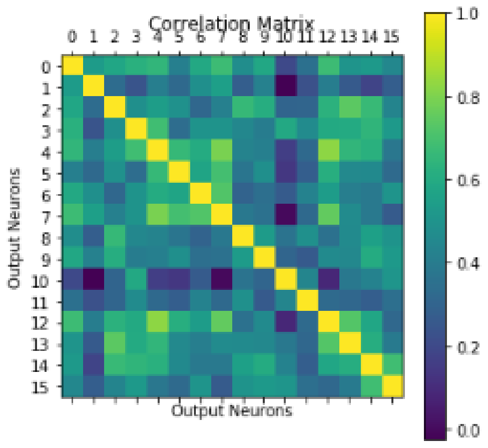


Table 1. Performance difference between when the standard competitive learning algorithm is used and the algorithm is applied by optimisation skills mentioned for this project.

	Without any methods	With all methods applied
Seed 10	7 dead units	5 dead units
Seed 20	8 dead units	5 dead units
Seed 30	8 dead units	6 dead units
Seed 40	6 dead units	6 dead units
Seed 50	7 dead units	7 dead units
Seed 60	7 dead units	7 dead units
Seed 70	9 dead units	6 dead units
Seed 80	7 dead units	6 dead units
Seed 90	6 dead units	6 dead units
Seed 100	8 dead units	6 dead units
Seed 1000	5 dead units	6 dead units

REFERENCES

[1] Rehman, J. (2019). *Difference between batch and online processing systems*. [online] IT Release. Available at: http://www.itrelease.com/2014/07/difference-batch-online-processing-systems/ [Accessed 26 Mar. 2019].

[2] Chegg.com. (2019). *Chegg.com*. [online] Available at: https://www.chegg.com/homework-help/explain-difference-online-processing-batch-processing-provid-chapter-10-problem-7rq-solution-9780538481618-exc [Accessed 26 Mar. 2019].

[3] Introduction to neural computation. (2019). [ebook] Nine, pp.217~229. Available at: https://vle.shef.ac.uk/bbcswebdav/pid-3745176-dt-content-rid-18820741_1/courses/COM3240.A.196261/Introduction%20to%20Neural%20Computation%20Pages%20217-229%281%29.pdf [Accessed 26 Mar. 2019].

[4] Docs.microsoft.com. (2019). *Normalize Data - Azure Machine Learning Studio*. [online] Available at: https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/normalize-data [Accessed 26 Mar. 2019].

[5] Sutton, G. and Reggia, J. (1994). Effects of normalization constraints on competitive learning. *IEEE Transactions on Neural Networks*, 5(3), pp.502-504.

[6] Quora. (2019). *Why is it important to add noise to the inputs of a neural network?*. [online] Available at: https://www.quora.com/Why-is-it-important-to-add-noise-to-the-inputs-of-a-neural-network [Accessed 26 Mar. 2019].