University of Sheffield

Collective Report A

Jake Quinn Sturgeon

Lecturer: Prof. Eleni Vasilaki

Module Code: COM3240

A report submitted in partial fulfilment of the requirements
for the degree of Computer Science with a Year in Industry MComp

in the

Department of Computer Science

April 1, 2019

Question 1

This report explores a form of unsupervised learning called Competitive Learning on the dataset EMNIST, which contains 7000, 88 by 88 px characters from 'A' to 'J'. The report's model of competitive learning is to use a single-layered feedforward artificial neural network (ANN), which also implements a winner takes all (WTA) approach - only one neuron fires for one specific pattern - and a stable Hebbian learning rule to calculate the weight change. The network has the goal of clustering the data into related groups. This is through a series of weights that represent the centres of each respected cluster. However, as the data is closely related, which can be seen in figure 1, 10 perfect representations of each letter is unlikely, thus more cluster centres must be produced to capture each centre with purity.

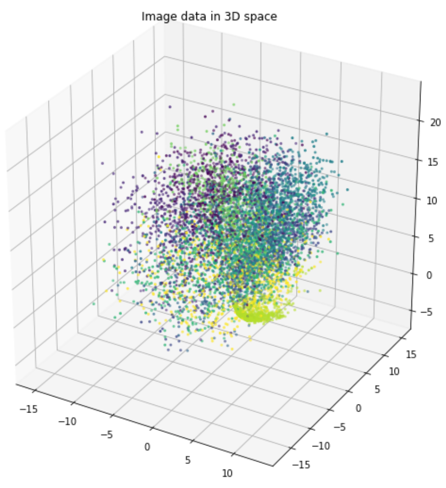

Figure 1: The EMNIST dataset represented in 3D space

Algorithm

1. Load and normalise the weights matrix, as well as set any model parameters and optimisations. Additionally, set a weight bias to the weights if this option has been set.

2. Randomly select a sample pattern, apply any presynaptic processing such as random noise or rotations if selected, and then followed by normalising the data.

3. Using equation 1. Calculate the postsynaptic weights and compute the winner neuron.

$$\boldsymbol{y} = \mathbf{W}.\boldsymbol{x}^T \tag{1}$$

```
# Activation function
y = np.dot(W, sample.T)
k = np.argmax(y)
```

4. Calculate the weight change by using the stable Hebbian learning rule 2. Compute a bias if selected.

$$\Delta \boldsymbol{w}_k = \eta(\boldsymbol{x}^\mu - \boldsymbol{w}_k) \tag{2}$$

```
dw = eta * (sample.T - W[k,:])
W[k,:] += dw
```

5. If selected, apply any secondary weight change to other non-winning neurons such as leaky learning and neighbourhood

6. Repeat step 2-5 until the end of the loop

The report's implementation uses an online learning variation of the algorithm, which is where learning takes place on each pattern and weights are updated sequentially. Another method is called batch learning, where learning occurs over the entire batch (group) of patterns in the dataset. During an epoch (a single run through the batch) the weights remain constant, while an error is calculated and added at the end of each epoch, usually with multiple epochs. Essentially, online learning uses a batch size of 1.

Normalisation

At initialisation the weight vectors are normalised such that $||\boldsymbol{w}_i|| = 1$ (snippet A.1) and each pattern is also normalised such that $||\boldsymbol{x}|| = 1$.

```
#Normalise sample pattern online
sample /= np.linalg.norm(sample, ord=2)
```

Normalisation is required as it solves stability problems, but also allows the use of the inner dot product to calculate the unit's activation function. Otherwise, the unit's activation function would be equivalent to a Euclidean distance calculation. This can lead to unstable and irregular convergences as large changes in the magnitude in each pattern vector would greatly affect a unit's winning chances, which could create a situation where a neuron starts to inadvertently dominate the network.
The report also uses the stable Oja's Hebbian learning rule 2. This rule is stable as subtracting the current weight from the sample pattern keeps a stable range of values. An example of an unbounded rule is 3 (Note, if the weights and samples are normalised to unit vectors after the rule is applied to a neuron, then 3 is a stable Hebbian learning rule)

$$\boldsymbol{w}_k = \eta \boldsymbol{x}^\mu \tag{3}$$

Question 2

Optimisation Techniques

Dead Neuron Metric

A dead neuron is a neuron that has barely learnt or hasn't learnt at all. The report's calculation of a dead neuron is a neuron that has won fewer times than a given percentage, which is calculated by the equation $\frac{1}{2} * \frac{1}{||\mathbf{W}||}$ (as **W** contains 25 prototoypes the percentage is 0.02). This method has much to be desired but usefully illustrates how constructive each neuron is to overall learning.

Cluster Purity

Secondly, cluster purity (equation 4) is a measure of the degree to which prototypes contain a single letter. However, this method uses the provided class labels to group each letter to a prototype, which isn't always available in unsupervised learning applications, but as this is just used as an evaluation of the method and not used in learning, it is still a valid unsupervised method. However, an increase of purity doesn't suggest that the network has learnt any better than a lower value of purity, as it doesn't penalise imbalanced data. For example, if we had only two prototypes for all ten letters and one prototype represented $N-1$ patterns and the other represent only 1 pattern. This means that the ANN hasn't effectively learnt anything but still gets a high purity value. It also doesn't penalise having a lot of clusters e.g., having a prototype for each pattern in the dataset. Therefore, it's only used as a prediction not fact that a neuron is converging to a cluster centre.
Each optimisation was tested over the seeds 0 to 4 to capture its variation, and an average of the number of dead neurons and cluster purity were calculated.

$$\frac{1}{N} \sum_{m \epsilon M} \max_{d \epsilon D} |m \cap d| \tag{4}$$

Control

The algorithm was run on seeds 0-4 with no optimisation techniques, which was used to create a baseline result (table 1). This baseline result will be used as a comparison of improvement for all the given optimisation techniques. The cluster purity shows that the baseline algorithm struggles to isolate each cluster effectively. It also shows that only a handful of neurons actually learn as around 18 out of 25 (72%) neurons do not learn.

Dead unit average	18.2
Cluster purity average	0.399

Table 1: Table of control results

Rotation

Rotation was used to rotate the handwritten images about its centre, which can be thought of as a form of normalisation as the algorithm tries to fit all the variations of rotation into one value. Shown in the code listing A.2. Each rotation degree was tested over seeds 0-4. A dead unit and cluster purity average were calculated. The results in table 2 show that the best rotation limit is (±°) 2°as it produced the best cluster purity as well as the lowest dead unit average. This leads to a reduction of 0.2 in dead units and a small increase of 0.005 for cluster purity. The increase in cluster purity is because some patterns are being rotated enough to move from one cluster space to another and due to the decrease in dead units, shows that this rotation is beneficial to the system. However, the system is still not effective as 18 neurons still not effectively learn.

Rotation(±°)	0	1	2	3	4	5	6	7	8	9	10
Dead units	18.2	18.4	18.0	18.4	18.4	18.8	18.0	18.2	18.2	18.8	19.0
Cluster purity	0.396	0.395	0.404	0.399	0.392	0.394	0.391	0.393	0.394	0.399	0.395

Table 2: Rotation results

Presynaptic Noise

Presynaptic noise is used to prevent neurons from reaching a poor local minimum, as noise can lead to a vector to escape its local minimum and continue its gradient descent. It additionaly, increases the winning chances of some dead neurons, as a dead neuron has very noisy weights, to win and update its weights. Each rotation limit coefficient was tested over seeds 0-4 and the implementation can be seen in the listing A.4. In the table 3 it is shown that as the noise limit increases, so does the cluster purity with the best being 0.08. However, the best dead unit average occurs when the noise limit is 0.03. 0.08 was chosen as this has a better cluster purity.

Noise	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
Dead units	18.4	18.2	17.6	18.0	17.8	18.0	18.0	18.2	18.0
Cluster purity	0.397	0.397	0.400	0.410	0.417	0.426	0.426	0.428	0.427

Table 3: Table of Noise results

Winner Bias

This optimisation (equation 5) is used to reduce the chances of a neuron continuously winning by applying a bias every time it wins. This allows neurons to start winning after a neuron has sufficiently converged to a cluster centre. Code can be seen in snippet A.5. The table 4 shows that the winner bias and dead neurons are negatively proportional. However, the average cluster purity also decreases as the average dead neurons decrease. This can be interpreted that a large bias causes prototypes to stop winning prematurely, which leads to other neurons to essentially learn something that they shouldn't. Therefore, increases cluster overlap thus, decreasing the average cluster purity. 0.1 will be chosen as this has the best cluster purity.

$$b = c * (\frac{1}{||\mathbf{W}||} - \frac{k}{t}) \tag{5}$$

$$\Delta \boldsymbol{w}_k = \eta((\boldsymbol{x}^{\mu} - \boldsymbol{w}_k) + b) \tag{5}$$

Winner Bias	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Dead units	6.2	4.8	4.2	4.4	4.2	4.4	3.2	3.6	3.6
Cluster purity	0.520	0.497	0.514	0.500	0.510	0.514	0.447	0.506	0.418

Table 4: Table of winner bias

Weight Bias
This technique is used to set a weight bias to each prototype where a neuron is set to a random pattern and then normalised (see listing A.3). Effectively setting each neuron to a cluster centre at initialisation (see snippet A.3), which removes most of the risk of having a dead neuron, as each weight vector will pointing towards a cluster centre. Therefore, this will greatly increase the chances all neurons to win/learn. This experiment was run on seeds 0 to 4. As expected, table 5 shows that there are zero dead units and a cluster purity of 0.537. Making the initial weight bias the best optimisation in this report. However, this method is not perfect, there is still a risk that each weight vector could be set to the same class of letter, which would lead to a very localised outcome.

Dead units	0
Cluster purity	0.537

Table 5: Table of Initial Weight Bias

Leaky Learning
Leaky Learning is used to apply noise to non-winning weights, which moves the losing weight vectors slightly towards a cluster centre. This increases the winning chance of losing neurons and can lead to less dead neurons. Code can be seen by the snippet A.6. Unfortunately, the table tab:leaky shows that leaky learning is actually detrimental to the system as it creates more dead units. The cluster purity was also negatively impacted by this method as it only produced poor results compared to the control. Therefore, 1e-05 was chosen as the optimisation as this had the least damaging effect on the system. A further experiment would be to explore if a smaller leaky value could improve the system's performance.
All Methods
In this section, all methods were applied to the seeds 0 - 4. As expected, table 7 compared to table 1 shows

Leaky Coefficient	1e-05	2e-05	3e-05	4e-05	5e-05	6e-05	7e-05	8e-05	9e-05
Dead Unit	18.6	18.2	18.8	19.0	18.6	18.8	18.4	18.4	18.8
Cluster Purity	0.396	0.392	0.388	0.385	0.379	0.372	0.370	0.367	0.356

Table 6: Table of leaky results

a massive increase in cluster purity (nearly 20%) as well as a drop to zero dead neurons. However, compared to table 5, it is only a small increase. This suggests that introducing noise into the system only provides marginally better results or two, optimisation should take each optimisation's interaction with the system into account. Therefore, further investigation would be to optimise all of the methods at once, rather than one by one, to see if this has greater affect on the system.

Dead units	0.0
Cluster purity	0.539

Table 7: Table of all. Ran with seeds 0 to 4

Secondly, figure 5 shows the number of times a neuron wins in the All Methods set up. When this is compared to figure A.4, it clearly shows that the control set up performs much more poorly. The figures 4 and A.1 also clearly show the increase of constructive learning these optimisations produce.

Question 3

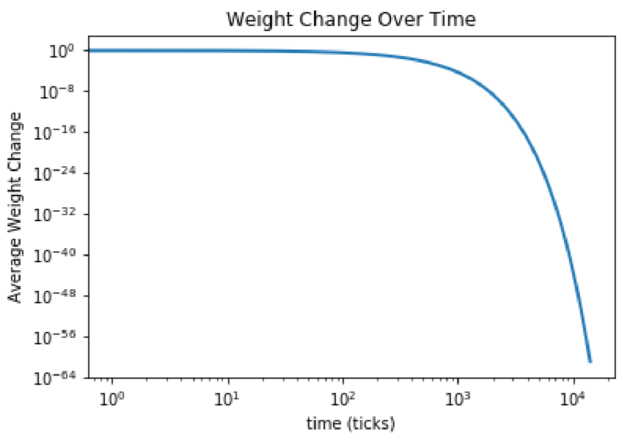

Figure 2: Running average (All methods)

A network can be said to have 'learnt' the features in the data when the weight change has converged to zero. As the weight change represents the gradient descent of the error function, then reaching a gradient of 0 shows that the network has reached a stable local minimum. As the algorithm is online, a moving average can be calculated. This is shown be the figure 2, which represents the weight changes (with a smoothing factor of 0.99) over time. The graph shows that the system has sufficiently clustered the data between 2000 and 10000 iterations as it shows a drop in magnitude of an order of 8. After this, the weights still converge to a stable value, but nothing significant will be learnt. Figure A.5 shows the control's weight change to converge around the same time.

Question 4

Prototypes

The system returns 25 prototypes (a postsynaptic neuron), which can be seen in Figure 4 . Each weight vector is represented by 7,744 dimensions, which means that each vector can be shown as a 88 by 88 px image. A prototype can be considered as a cluster's centre or a mean of all the added weights. This can be shown by the third prototype, which clearly resembles the letter E. However, some prototypes could be described as being the centre of two clusters, or of not learning at all, which can be shown by the top left prototype looking like a combination of the letter G and F. Figure A.1 shows each prototype under the control conditions and has a lot of noisy images. These are considered to be dead neurons as they have learnt very little, which can be further shown by figure A.4 as these neurons only win a couple of times.

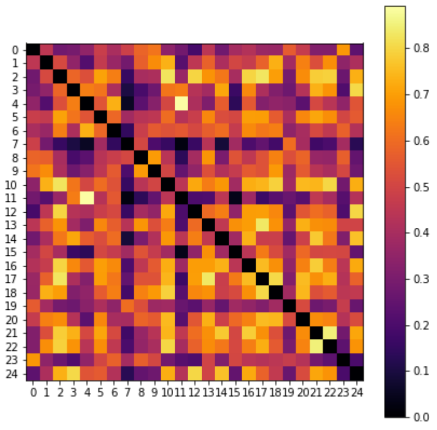

Figure 3: Correlation Matrix (All methods)

Correlation matrix

Figure 3 is a visual representation of a correlation matrix of the prototypes in 4. Where the brighter the pixel, the greater the correlation between the two prototypes. It can be seen that a lot of the prototypes correlate. Thus, by using the correlation coefficients, an algorithm could be developed to combine highly correlating prototypes together. However, each prototype contains a lot of empty space and most of the 'important' data is located in the middle of the image. Therefore, a lot of prototypes will have a large correlation coefficient with each other, even if they are different letters, which would lead to two separate cluster centres becoming one and undoing the clustering algorithm's work. A future suggestion would be to use a dimensionality reduction technique such as PCA to remove some of the less useful dimensions.

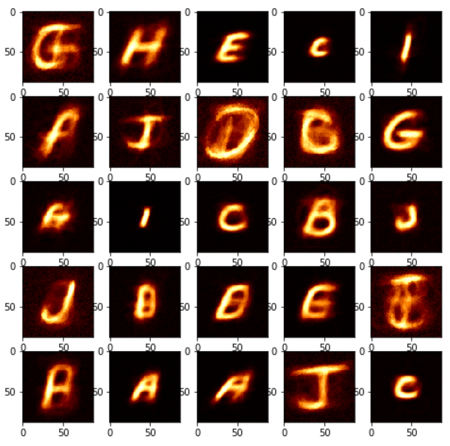

Figure 4: List of prototypes (All methods)

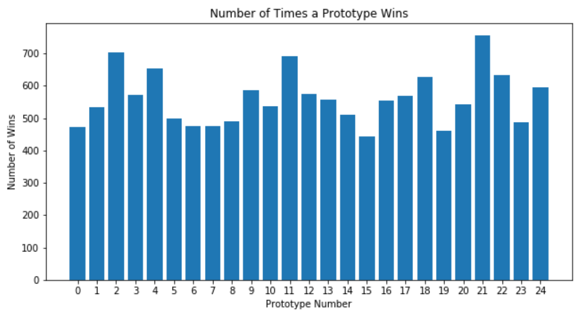

Figure 5: Number of times a neuron won (All methods)

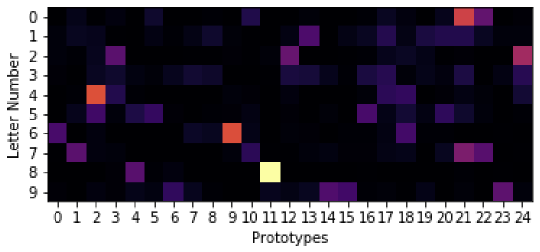

Figure 6: Matching matrix (All methods)

Furthermore, figure 6 (snippet A.7) is a matching matrix, which is used to visually represent the cluster's purity. Ideally, one bright pixel would be in one row as this would show that each prototype successfully contains only one class. It shows that prototype 11, which is an 'I', is a very pure cluster. Purity is closely related to the winning count as the bright pixels in figure 5 closely correlate to bright pixels in figure 6 .

Appendix

```
# init and normalise weights
W = winit * np.random.rand(digits,n)
normW = np.sqrt(np.diag(W.dot(W.T)))
normW = normW.reshape(digits,-1)
W = W / normW
```

Listing A.1: Weight Initialisation

```
def rotation(sample, rotation_limit):
    """
    This function rotates the pattern within a given range

    """
    # Get a random angle within the range +- rotation_limit
    angle = np.random.uniform(-rotation_limit,rotation_limit)
    sample = rotate.rotate(np.reshape(sample, (88,88)), angle, reshape=False)
    sample = np.reshape(sample, n) # Back to a 1 by n vector
    return sample
```

Listing A.2: Rotaion Optimisation

```
def init_bias(W):
    """
    This function returns a biased ANN where each prototype
    is set to an example

    """
    #  For every prototype
    for i in range(0,digits):
        # Set the normised sample as the neuron w
        r = np.random.randint(0,m)
        sample = train[:,r].copy()

        sample = sample/np.linalg.norm(sample,ord=2,axis=0)
       W[i,:] += sample
    return W / np.linalg.norm(W, ord=2, axis=0)
```

Listing A.3: Weight Bias Optimisation

```
sample += noise_limit * np.random.rand(n)
```

Listing A.4: Noise Optimisation

```
def bias(c, k, counter, eta, x, W, t):
    """
    This function returns the bias adjust weight
        for a given prototype

    c -- Prototype bias coefficent
    counter -- ndarry containing the number times a
        prototype has been 'hit'
    eta -- Learning rate
    x -- sample pattern
    W -- ANN

    """
    b = c * ((len(W)**(-1)) - counter[k]/t)
    return eta * ((x - W[k,:]) + b)
```

Listing A.5: Weight Bias Optimisation

```
W[:k,:] += (sample.T - W[:k,:]) * leaky
W[k+1:,:] += (sample.T - W[k+1:,:]) * leaky
```

Listing A.6: Leaky Learning Optimisation

```
def matching_matrix(W, digits):
    """
    Produces a matching matrix

    """
    match = np.zeros((len(set(trainlabels)), digits))
    for i in range(0,m):
        sample = train[:, i] / np.linalg.norm(train[:, i], axis=0)
        vals = W @ sample
        match[trainlabels_nums[i], np.argmax(vals)] += 1
    return match
```

Listing A.7: Matching Matrix

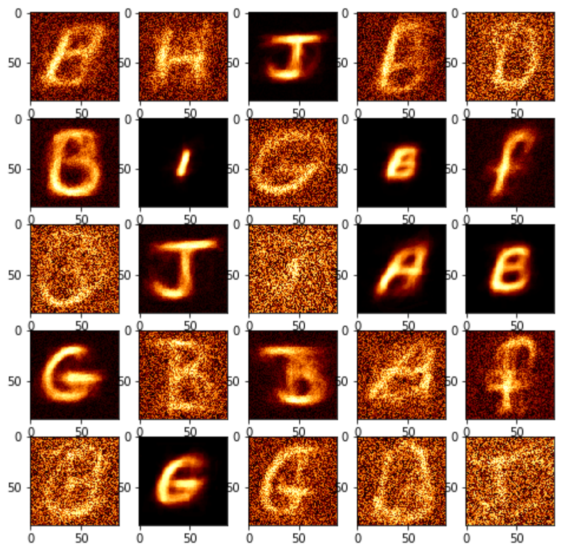

Figure A.1: List of prototypes (Control)

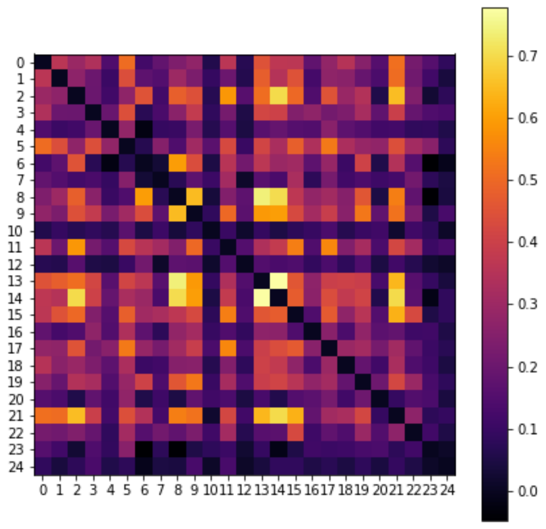

Figure A.3: Correlation Matrix (Control)

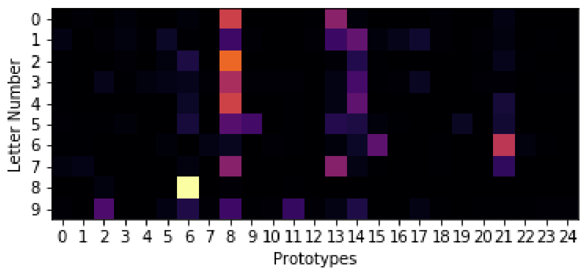

Figure A.2: Matching matrix (Control)

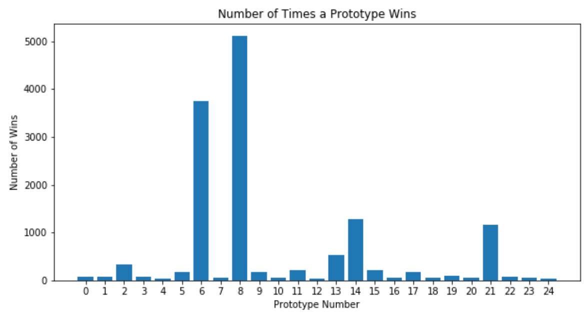

Figure A.4: Number of times a neuron won (Control)

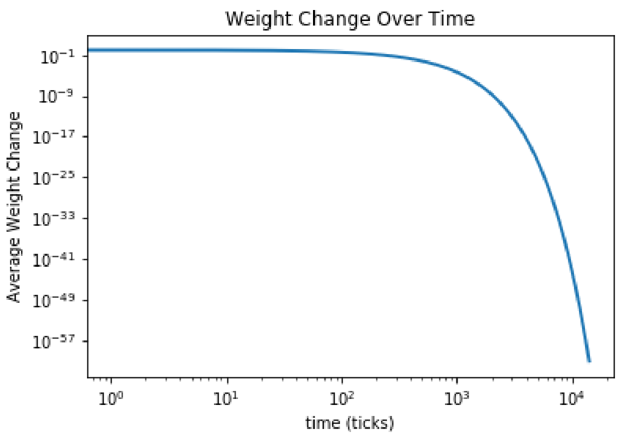

Figure A.5: Running average (Control)