Assignment COM3240

Competitive Learning

2019

Simon Drake

Student number: 160152089

1. Introduction

In the context of machine learning using neural networks has proven to be extremely useful in many circumstances. Many supervised, semi-supervised and unsupervised techniques have been explored in their advantages and disadvantages. I have focused on an unsupervised approach, specifically an online competitive learning algorithm to detect clusters in pixel data and classify numbers. I have gone on to show that given some data and the right parameters a one-layered neural network can effectively find the clusters in the data. Moreover given a set of inputs we can optimise the search algorithm to spread out its prototypes.

2. Algorithms

2.1 A competitive learning algorithm

The distinction between a competitive learning algorithm and other unsupervised learning algorithms is that only *one* output neuron is activated, or *one* per group. These output units are often called “winner-take-all” units. The purpose of this type of network is so that output neurons are tuned to certain inputs that are similar. This allows us to categorise the input data into the same class when they fire the same output neuron.

2.2 Batch and online learning

Batch and online are two approaches one could take in writing an algorithm which processes input and updates the weights after some iteration. The difference is that the batch algorithm keeps the weights between neurons constant while computing the error associated with each instance or a group of instances in the input. The online approach would update the weights of the system after each individual instance is seen. The batch algorithm is slightly more efficient in terms of computations but the path to the minimum (the path drawn out by gradient descent) is not as defined as an online algorithm. Most algorithms are batch.

2.3 Dataset

Our experiments are performed on the EMIST data set which contain 7000 training examples of the letters A-J, the classes are balanced so there are 700 instances of each letter. Each instance is a construct of 88x88 pixel values.

2.4 Our algorithm

The algorithm we have implemented is threefold. In it’s simplest form it computes the change in weight, **ðw***, of a winner for each iteration in a decided number of iterations and updates the weights accordingly. As we found that outputs often converged to the same area in a cluster we extended this algorithm to, after some step, compute either the distance between weight vectors (extension 1.1) or the *Pearson product-moment correlation coefficients* (extension 1.2) *a*nd re-initialise weights that were too close/similar to one being looked at. Our results will show experiments with and without these extensions.

2.4.1 Core algorithm

Initially we chose an input instance, **ʒ**, randomly and compute the output of each neuron by computing the dot product of the weight vector and the input vector:

$$\mathbf{w} \cdot \mathbf{ʒ} = \mathbf{O}$$

The "winner" is taken by choosing the highest activation value:

$$\mathbf{O_n = max(O)}$$

The weights connected to this output neuron, **w***, are updated by adding a change calculated by multiplying a learning rate **ŋ** with the distance between those weights and the input instance, **3:**

$$\mathbf{new\ w^* = old\ w^* + ŋ(3 - w^*)}$$

This process repeats for **x** iterations and then exits (See appendix A). It is important that the weight vectors and the input vectors are normalised or taking the difference between the input vector and weight vector would not give us the results we want.

2.4.2 Extended algorithm

Extension 1.1 makes use of the properties of the vector space, in that the magnitude[1] of the vector obtained by taking the difference between two weights will be small if those vectors are close to each other in vector space. Thus a matrix of all the lengths of the distances between all the weights is calculated:

$$\mathbf{len\_distance(w_i, w_j) = len(w_i - w_j)}$$
$$\mathbf{M_{ij} = len\_distance(w_i, w_j)\ (for\ all\ i\ and\ j)}$$

If we decide to use the correlation matrix:

$$\mathbf{M_{ij} = Pearson\ product\text{-}moment\ correlation\ coefficient(w_i, w_j)\ (for\ all\ i\ and\ for\ all\ j)}$$

Once this matrix is computed we compare each element with every other element, if the distance between $\mathbf{w_i}$ and $\mathbf{w_j}$ is below some threshold value or the correlation is above some threshold value then a new weight is randomly generated for $\mathbf{w_j}$ which is normalised and goes to replace the old one. To be sure that the weight is not changed again it is added to a set "*changed_weights*". (See Appendix C)

This process is repeated only after the algorithm has had enough time for the weights to stabilise and only if the re-initialised weight has had enough time to stabilise (after some step in iterations). If *changed_weights* is empty, all the distances between the vectors are above the threshold value or all the correlations are below the threshold value and we can exit.

3. Experiments & Results

3.1 Avoiding dead units

One common problem with competitive learning is that a weight vector is so distant from any input that a particular output never activates. These outputs are called "dead units". Similarly, weights which only win a small percentage of the time are not very useful. Thus a "best-case" scenario is when the outputs win a similar amount of times. Our first experiments explores the relationship between the number of winners and the number of iterations and the learning rate. We have implemented a *counter* which counts the number of times an output wins. For **n** outputs the length of the *counter* will be **n**. Since our dataset contains balanced classes and the probability of picking any letter is evenly distributed, as a metric we have chosen the variance of *counter* to see if there is a large imbalance between counts

1 I use the term "magnitude" and "length" interchangeably when referring to vector quantities.

Number of iterations	Variance(counter)
5000	0.14×10^6
10000	2.10×10^6
20000	7.24×10^6
40000	55.02×10^6
80000	251.97×10^6

Table 1: Variance of the counter at a learning rate of 0.05 and 10 outputs.

As we can see from the Table 1 above given a constant learning rate of 0.05 the variance of counter increases as the number of iterations increase i.e some weights win much more often than others. The number of iterations and the variance of the counter are positively correlated. Next, we look at the effect of the learning rate on the variance of counter for a constant number of iterations.

Learning rate	Variance(counter)
5×10^{-1}	0.97×10^6
5×10^{-2}	8.43×10^6
5×10^{-3}	2.09×10^6
5×10^{-4}	0.29×10^6
5×10^{-5}	0.24×10^6

Table 2: Variance of the counter with 20 000 iterations and 10 outputs.

As we can see from the Table 2 above given a constant number of iterations the variance of counter decreases as the learning rate decreases. This exercise is much more volatile than the last and sometimes results in anomalies. However the general trend is constant among test runs: the learning rate and the variance of counter are negatively correlated.

3.2 Impact of the ðw* vector

The impact that **ðw*** may have on a particular weight is important to look at as it allows us to see at which stage it may be appropriate to stop the loop. To investigate how the change in weights behave we have plotted their magnitude against the number of times they have been computed for every weight vector. i.e. for 10 outputs we have 10 graphs with magnitude on the y axis and number of times that weight has been changed on the x axis. We ran our core algorithm with 100 000 iterations and a learning rate of 0.0025 and an extended variation of this. We can see that the general trend is towards the weight changing significantly in the first 1000 iterations (first 1000 for *that* weight not global number of iterations). It is important to note here that this is not the case for our extended algorithm if the weight is re-initialised then the magnitude of **dw*** will rise and go through a steep drop again. (See appendix D[2])

3.3 The number of iterations

It is also worth noting that the number of iterations has an impact on the convergence of the prototypes, given a learning rate the prototypes being initialised randomly will find separate clusters quite well but given enough time they will converge to a common overlapping cluster in out case usually “I”. The “I” shares pixels with almost all other letters and so prototypes that have converged to other letters

2 All our appendices showing the result of a run will have the command used.

will eventually fall into the I cluster. You can find an illustration of this in Appendix E.

3.4 Number of prototypes

Generally, due to the way the algorithm works the more prototypes you have the more chances you have at having one prototype reach a particular cluster of data. Moreover it allows us to make sure we “cover” one particular cluster, for instance the letter “i” written slanted to the left and one slanted to the right, will only actually over lap in a small fraction of pixels. Thus it is important to have both writing styles covered. This is illustrated in Appendix F. Again it is important to have a low learning rate so the prototypes don’t over-converge. Too many prototypes however and you lose specificity.

3.5 Extension in action

In the interest of visualising how the algorithm benefits from the extensions we have shown in Appendix G what the prototypes look like before a re-initialisation and afterwards. As you can see for G.1 the 8th prototype looked a lot like the 4th and for G.2 the 5th looked like the 1st . This demonstrated that the algorithm is doing as intended and can prohibit prototypes from converging to very similar positions.

We will now look at the behaviour of our two extensions given certain parameters.

Threshold	Exit iteration
0.81	58 000
0.82	38 000
0.83	26 000
0.84	32 000
0.85	22 000

Table 3: Extension 1.2 called with max iteration 60 000, min iteration 30 000 and step 2 000

Threshold	Exit iteration
0.11	24 000
0.12	26 000
0.13	28 000
0.14	30 000
0.15	32 000

Table 4: Extension 1.1 called with max iteration 60 000, min iteration 30 000 and step 2 000

The lower the correlation threshold the more re-initialisations the algorithm goes through and the more iterations it takes to reach appropriate distances. Similarly the higher the threshold for the length of the distance vector the more iterations it takes to stabilise. This is illustrated in Table 3 and 4.

It is important to note that as we are choosing our training instances randomly and more importantly we are initialising our weights randomly there is a level of stochasticity involved. These results may not be directly replicated however general trends should be consistent among similar experiments.

4. Conclusion

The unsupervised, online, single-layered completive algorithm works extremely well to find clusters in data. Given a dataset or type of dataset one should experiment with the number of iterations and learning rate to achieve low variance and decrease the risk of dead units. The system seems to converge very quickly to a solution but steps can be taken to optimise the weight vectors during the convergence. If left too long weight vectors may fall out of their optimal position.

Appendix A: Core algorithm

```
# Get a randomly generated index in the input range
i = math.ceil(dt.m * np.random.rand())-1
x = dt.train[:,i]

# Get output of all neurons, and find the winner's index
h = dt.W.dot(x)
h = h.reshape(h.shape[0],-1)
output = np.max(h)
k = np.argmax(h)

# Calculate the change in weight for the winner and update it
dw = eta * (x.T - dt.W[k,:])
dt.W[k,:] = dt.W[k,:] + dw
```

Appendix B: Extension 1.1

```
# Returns a matrix of the magnitudes of the difference in weights for all weights
def get_distances(W):

    # For all weights:
    # Get the magnitude of the distance between weights
    distances = list()
    for i, item in enumerate(W):
        distances.append(list())
        for j in W:
            distance = np.linalg.norm(item - j)
            distances[i].append(distance)

    X = np.stack((distances), axis = 0)
    return X
```

Appendix C: Comparing distances or correlations

```
if "-w" in cl.opts or "-c" in cl.opts:
    # Print prototype of what the prototypes look like before any re-initialisation.
    if t == tmin:
        printprototypes(dt.W, cl.args[0], cl.args[1], t, threshold)

    if t > tmin and t % step == 0:
        changed_weights = set()
        if "-w" in cl.opts:
            X = get_distances(dt.W)
        else:
            X = np.corrcoef(dt.W)*-1

        for i, item in enumerate(X):
            for j, item_j in enumerate(item):
                # If we are comparing the same weights, or the weight has already been changed, move on.
                if i == j or (j in changed_weights) or (i in changed_weights):
                    continue
                else:
                    # If the distance is lower than the distance threshold required:
                    if item_j < threshold:
                        # Create a new weight vecotr and normalise it
                        new_weight = np.random.rand(1, np.shape(x)[0])
                        new_weight = new_weight/np.linalg.norm(new_weight)

                        # Replace the old weight vector
                        printprototypes(dt.W, cl.args[0], cl.args[1], t-1, randint(0,99))
                        dt.W[j] = new_weight
                        printprototypes(dt.W, cl.args[0], cl.args[1], t, randint(0,99))

                        changed_weights.add(j)

        # If no weights have been changes, distances are good and we can exit the while loop
        if len(changed_weights) == 0 :
            too_close = False
```

Appendix D: Core algorithm and extension 1.1

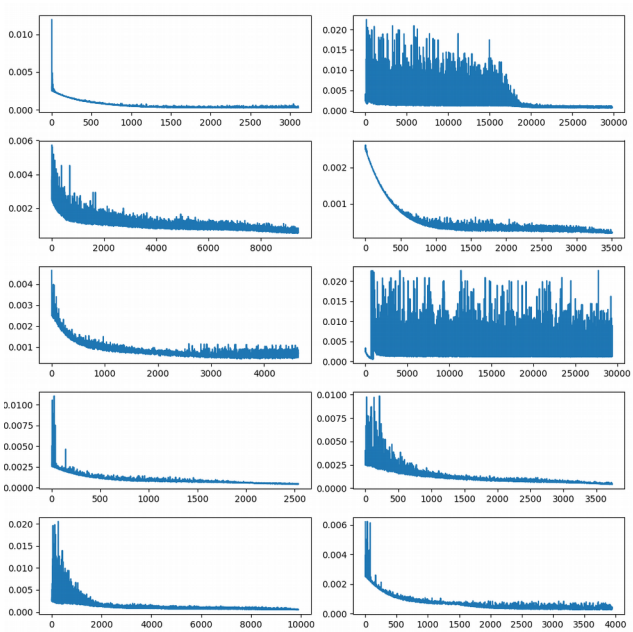


Illustration 1: python3 competitiveL.py 0.025 100000

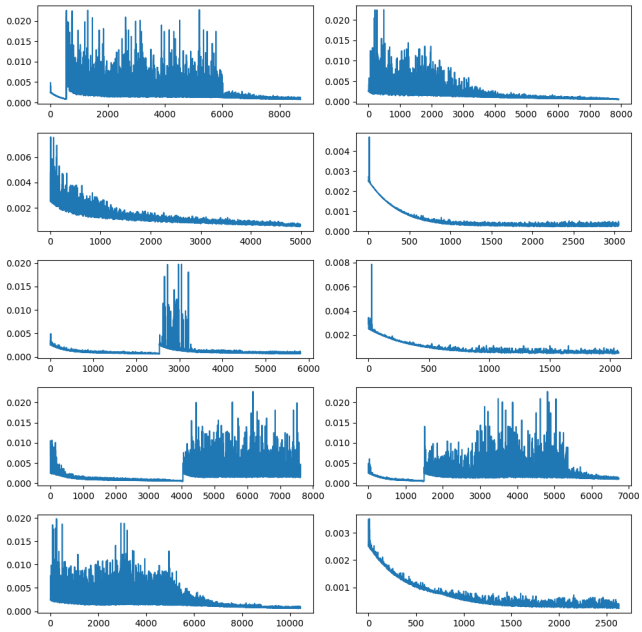
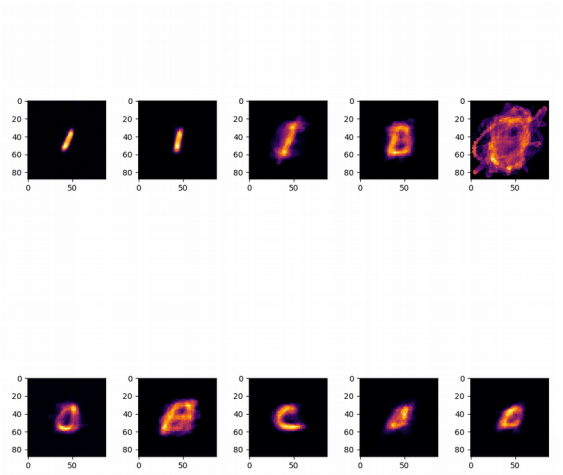
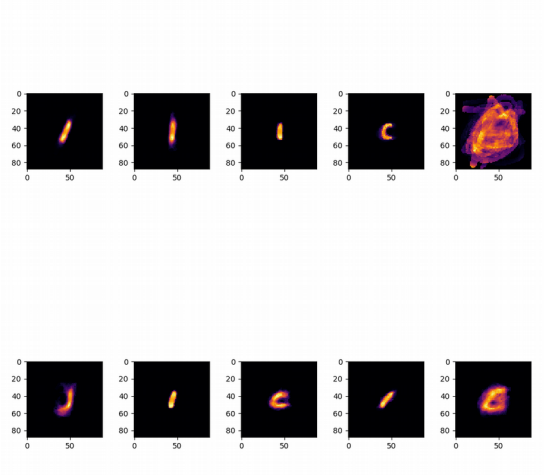


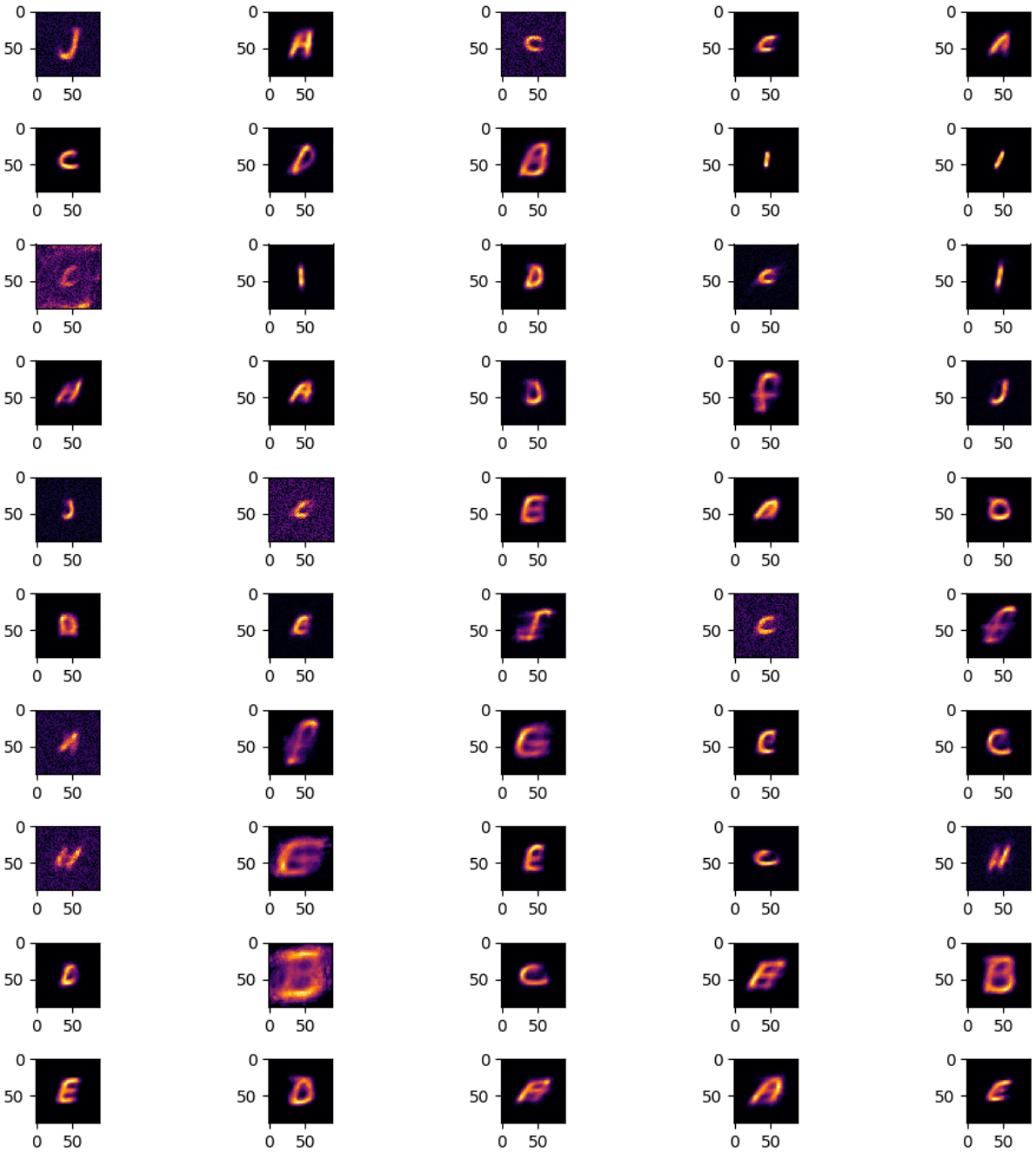
Illustration 2: python3 competitiveL.py -w 0.025 100000 30000 10000 0.09

Appendix E: python3 0.05 100000 (at 2000 iterations and 100 000 - uncomment lines 175, 176 to print the prototypes at 2000 iterations)

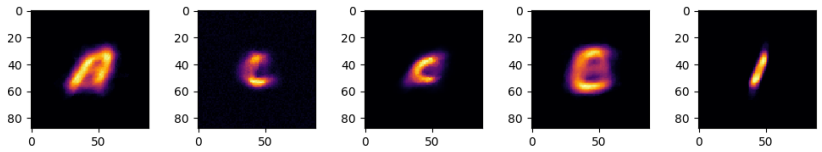


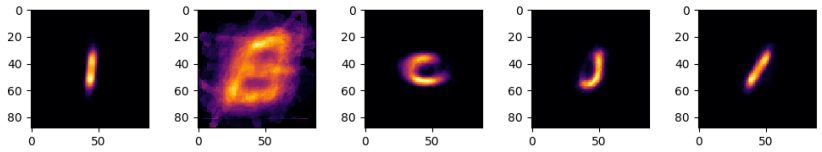


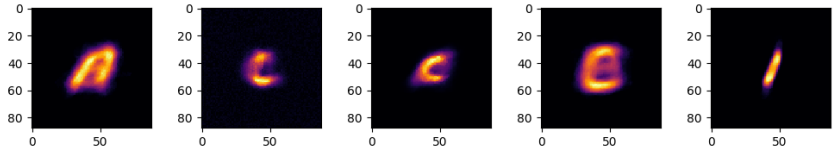
Appendix F: python3 competitiveL.py -w 0.008
80000 30000 10000 0.07 (50 outputs)

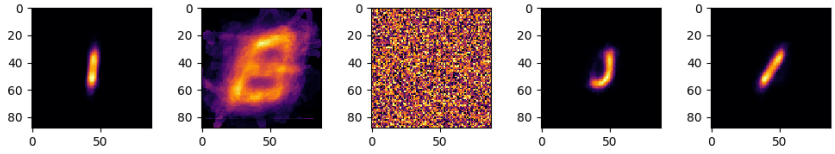


Appendix G.1 python3 competitiveL.py 0.008 50000 30000 10000 -0.91 (uncomment lines 199 and 200)









Appendix G.2 : python3 competitiveL.py 0.008
50000 30000 10000 -0.91