Competitive learning on EMNIST data-set

Vladyslav Bondarenko

I. Competitive Learning

A. Introduction

A simple competitive learning network consists of two layers, where the input layer ξ is fully connected to the output layer O through excitatory connections, also called weights. In theory each output unit represents a cluster which corresponds to a generalised version of one of the classes from the data-set. For example, ruining a competitive learning algorithm for EMNIST data-set should produce generalised version of different letters and with addition of extra units possibly further variation of the way those letters are written. The network learns via competition where usually, the highest output is selected representing the winner neuron and only weights leading to that neuron are adjusted accordingly.

B. Algorithm Implementation

1) Initialise and Normalise: First, all the parameters are initialised and the data set is prepared for the network. The EMNIST data-set consists of 7000 images of 10 different hand written letters A-J. Each letter is an 88*88 pixel image which has been converted to a one dimensional vector. For the purpose of competitive learning only the direction of the vector matters and thus it is important to normalise the data before training the network. This is because the output of the network is calculated via dot product between weight matrix W and input pattern ξ, where the dot product is given by:

$$W \bullet x = |W| * |\xi| * cos(\theta) \tag{1}$$

Thus if data is not normalised, the difference between magnitudes in two vectors can cause the larger vector to win the competition even tho it is pointing further away from the desired direction. This can lead to dead neurons as vectors with small magnitude may never fire. From (1) it is also easy to see that the weights W also have to be normalised. This is done straight after the random initialisation of the weight matrix where rows correspond to the output unit and columns to the input. The value W_{ij} is the weight of the connection between output neuron *i* and the input neuron *j*.

2) Standard Competitive Learning: Next, the algorithm enters a loop where an input vector ξ is chosen at random from the data-set and fed trough the network producing output h_i:

$$h_i = \sum_i W_{ij}\xi_j = W_i \bullet \xi \tag{2}$$

The maximum output *i** is chosen and change in weight for the according neuron is calculated using the standard competitive learning rule [1]:

$$\Delta W_{i*_j} = \eta(\xi_j^\mu - W_{i*_j}) \tag{3}$$

This learning rule is derived from the Oja's rule and so with normalised data the weights should converge at a size of 1 after a number of iterations. At this point, with favourable conditions, the weights would move towards samples in the data-set and represent the average data point of respective clusters.

3) Batch vs Online: The algorithm above is an example of an online learning, because the weights are updated after each iteration. This is different to batch algorithm where the weights are kept constant while the error associated with each sample in the input in computed. Only at the end or between the epochs the weights are updated. Although those are different approaches and produce different behaviours, they both converge at a same minimum due to parabolic nature of the error function. However, with addition of noise, online learning approach gains an opportunity to get out of local minimum due to is susceptibility to noise. This has to be used with caution as too much noise can lead to miss-representation of output units as it starts jumping between different clusters.

II. Optimisation

Before moving onto discussing optimization techniques it is important to look at what is being optimized. This is a relatively controversial topic as there is no single value to optimize within competitive learning. A number of measures were considered for this task which are described in detail in following few paragraphs.

A. Measures of Improvement

1) Dead Units: A common problem in competitive learning is occurrence of dead units and thus reducing dead units is considered as one of the more appropriate optimization techniques. A dead unit is usually defined as the unit that doesn't win often enough and thus it lies far away from any character vector without a chance to move closer. It has lost the competition and will continue loosing and so becomes useless for the network. From this, it is common to label output units as dead if it has a low firing rate at the end of the epoch. However, Fig. 1 shows 4 different neurons run for same number of iterations but at different learning rates: smaller to larger if going from left to right. Thus for example, Neuron 1 fires 1026, but it is obvious that it has not moved close enough to any character. Furthermore, because the weights start to converge, it is possible that even with additional iterations and more wins for this neuron it will not move close enough to any cluster. Following this logic it can be classed as a dead unit.

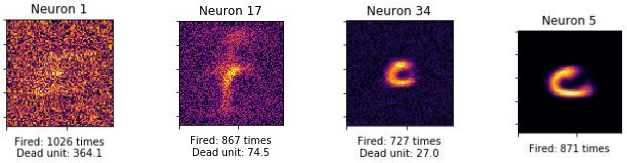


Fig. 1: Dead unit progression

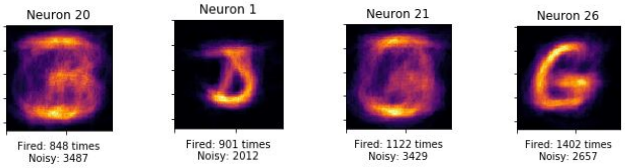


Fig. 2: Distorted Units

Learning R	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08
#Dead	4.7	1.2	1.1	0.7	0.3	0.2	0.0	0.2
#Correlated	4.8	19.9	41.3	48.6	51.7	66.6	64.5	65.5
#Distorted	4.4	2.4	1.2	1.2	1.0	0.6	1.2	0.2

TABLE I: Optimal Learning Rate for 36 output units

Thus, a measure for dead units was established, which calculates the noise within the image rather then the number of times a unit has fired. Noise was calculated using a method described in [2]. The output of the calculation is a float above 1 as shown in Fig. 1, where for Neuron 1 noise level is 364.1 and for Neuron 34 it is 27. It was found that image becomes clear at a noise level of 10 and so everything above that value is classed as a dead unit.

There is an obvious flaw for this measure where for example too low iteration number or too small learning rate will cause a lot of dead units, even tho given more time they would learn. This can be avoided by setting a reasonable learning rate and number of iteration. In fact, in this experiment an average firing rate was kept constant and number of iterations adjusted accordingly to match number of output units.

2) Correlation: Although dead units are a good and reliable measure of performance, there are a number of other important aspects to consider. One of them is the correlation between output units. It is no use if the network has 0 dead units, but all the output units are the same. It is important to encompass all of the variations in data that are present in training set and thus, less correlated output units is a desirable outcome. The details of correlation calculations its impact will be described in a later section.

3) Distortion: Another situation that occurs often is a unit that during learning jumps from cluster to cluster or overgeneralises for a given letter. An example of this is presented in Fig. 5. The distortion is measured very roughly, where a number of pixels that have above average value is recorded and based on that distortion is determined. Units with over 2000 pixels with values above average are classed as distorted. This proven to be a relatively good estimate identifying most of the misrepresenting units, including a common problem between clusters for letters A and B due to their similarity and a large variation of the way they can be written. Another connection was notices where it was common for neuron that fire too often to become distorted. However, there is a disadvantage with this approach as larger letters could be regarded as distorted even tho they just take up more space than average. This is also a case for very small letters which even tho distorted, are not classed as so. Therefore, this measure will be given a lower priority when considering final results.

B. Optimization Techniques

The main focus of optimisation is to reduce the number of dead units while keeping the other two measure relatively stable. This will be done using a number of approaches described in [1]:

- Leaky learning - updating the weights of both winners and losers. Losers weights are updated with a much smaller learning rate. This should allow loosing outputs to slowly move towards average input direction and eventually start wining the competition.
- Conscience - involves a bias term μ which is relate to the number of times a neuron i fires. It should increase if unit fires too often and decrease otherwise. This bias is then subtracted from output h. Precise algorithm for this was taken from [3].
- Noise - using a long-tail distribution to smear the pattern vector. This should give some small probability for any unit to win the competition and thus a possibility for a dead unit to fire.
- Decaying learning rate - although not directly aimed at reducing number of dead units, this approach should in theory improve the performance of the network. Specifically, the number of distorted neurons may reduce due to the fine tuning that occurs as the learning rate is reduced.

C. Parameter choice

All the parameters were chosen in a similar fashion to how the optimal learning rate was investigated in Table I. A random value was guessed and then adjusted to improve the values of performance measures. In case of learning rate 0.01 was chosen because increase to 0.02 cause a slight decrease in dead and distorted units, but a proportionally larger increase in number of correlated units. Those parameters are unlikely to be the most optimal, however, full investigation is computationally demanding and time consuming and is regarded as beyond the scope of this project.

It is important to note that for the decaying learning rate configuration, initial learning rate was set to a very high value of 0.9. It was found that by setting a learning rate to 1 without decay the network would over-fit to represent precisely one of the data points from training set. By setting high initial learning rate with decay, we essentially add another technique described in [1], which involves initialising weights to randomly selected inputs from the training data. This ensures that all the weights are representing a valid cluster event if they rarely win the competition as well as benefits from the low learning rate at the end for fine tuning.

# Output Units:		10	16	25	36	49	64	81	AVERAGE
Standard Competitive	% Dead units	6	7.5	8.4	11.2	12.7	14.1	14.1	10.6
	% Correlated	8	4.4	20.8	36	58.6	88.0	120	48
	% Distorted	23	21.3	20.4	16.9	15.7	13.3	13.8	17.8
Leaky Learning	% Dead units	8	8.75	7.6	12.5	10.8	13.9	13.6	10.7
	% Correlated	2	4.3	19.2	37.7	63	95	116.8	48.3
	% Distorted	23	22.5	18.8	16.1	15.3	14.2	15.2	17.9
Conscience	% Dead units	2	5.6	7.2	11.9	10.6	12.6	13.7	9.3
	% Correlated	3	6.9	16.8	37.2	63.8	93.1	126.4	49.6
	% Distorted	29	26.3	18.4	19.4	16.9	15.9	14	20
Noise	% Dead units	2	7.5	5.6	4.4	3.3	2.7	1.5	3.9
	% Correlated	8	9.4	25.6	48	91.2	162	260	86.3
	% Distorted	24	22.5	18.4	17.8	17	15.5	15.9	18.7
Decay learning	% Dead units	2	3.1	7.6	10.3	18.5	22	27.1	12.5
	% Correlated	4	13.8	24.8	37.8	42	52	60.6	33.6
	% Distorted	13	14.4	14.8	16.9	16.9	18.6	17.7	16
AVERAGE	% Dead units	4	6.5	7.28	10	11.2	13.2	14	
	% Correlated	5	7.8	21.5	39.3	63.7	98	136.8	
	% Distorted	22.4	21.4	18.16	17.4	16.4	15.5	15.3	

TABLE II: Table of Results

D. Results

For each optimization technique an experiment was conducted to find the proportion of dead units, correlated units and distorted units to the total number of output neurons. The average firing rate was kept stable at 1000 throughout the experiment and learning rate at 0.01, except for when decay learning was tested. Each configuration was run 10 times, the average was obtained and recorded in Table II. Further averages were calculated to represent the general results for different approaches and different number of output layers. Based on those results, the parameters were slightly adjusted and the most optimal configuration chosen.

From that table we can see that if the aim is to minimise the number of dead units then noise is by far the best approach with only 3.9% dead units. However, the average proportion of correlated units is also extremely high which suggests that the noise was too high. This would cause neurons to fire randomly, generalising to the average point of the whole data-set rather than of individual clusters. This is where the measure for distortion fails, because units are not representing valid letters from data-set but distortion is still very low. Reducing the noise slightly should result in a more balanced output.

On the other hand, decay learning had a high count of dead units of 12.5% but at the same time the lowest correlation count and distortion. In theory combining those two approaches could produce well balanced results. It is also worth observing that for networks with smaller number of output units the proportion of dead units is much smaller. Therefore, decay learning is an optimal approach for smaller networks. It is likely that the number of dead units starts to grow because all of the potential clusters have been found.

Conscience approach has reduced the number of dead units slightly at a cost of distorted units. Although not so effective in this experiment, this approach is very promising. It is mentioned in [1], how challenging it is to achieve stability while using this approach and so more work would have to be done to ensure that the most optimal parameters and equations are used. Moreover, the algorithm was slightly adapted from [2] and so would have a number of flaws.

Leaky learning has almost no impact on the performance of the network which is most likely because of an extremely low parameter. However, increasing that parameter causes even worth results and therefore, leaky learning will not be used in the final configuration. Surprisingly, even the standard competitive learning algorithm without any additions performed well and had a comparatively balanced results.

In terms of number of output units, it is clear that as it increases both proportion of dead and correlated units increases. At the same time proportion of distorted units decreases. Thus to keep a balance, 36 output units were chosen for the final experiment. Overall, the final experiment was ran with the combination of decay learning and conscience algorithm with slightly adjusted parameters. The results of this experiment will be discussed further down along with prototypes and correlation matrix.

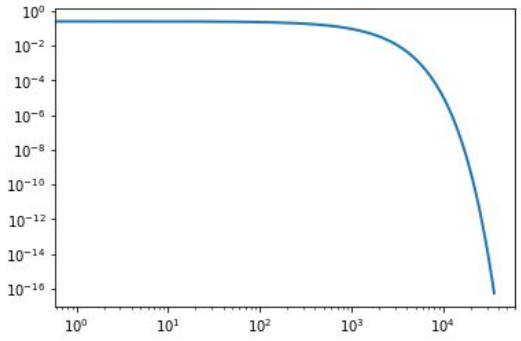
III. Weight Change

The weight change over time was recorded for each configuration and plotted on a log-log scale. Fig. 3 shows at (a) the weight change over time for constant learning rate of 0.01. Because the average firing rate was also kept constant, (a) is a representative of the change in weight for all configurations with that number of dead units. It is important to note that with higher number of prototypes there are more iterations and therefor average weight curve reaches much lower values (b). Due to noisy nature of online learning, a smoothing technique was used to produce a continuous curve. Is is clear that after $10^4 - 10^5$ iterations the change in weight starts to decay with increasing speed and so it is assumed that the network has learned. Continuing the learning process would produce extremely small changes in weights and so there will be almost no effect on final result.

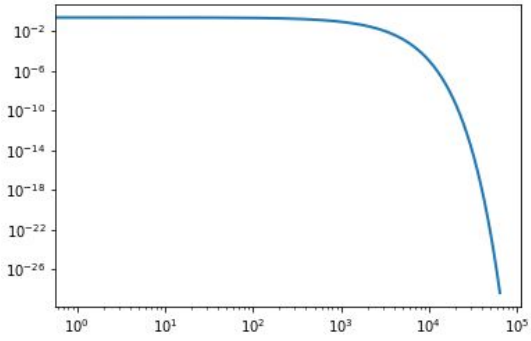
IV. Prototypes and Correlation

A. Prototypes

As can be observed from Fig.10 the network was able to find a number of prototypes where some of them are distorted



(a) 36 Output units



(b) 64 Output units

Fig. 3: Weight change over time

Fig. 4: Prototypes

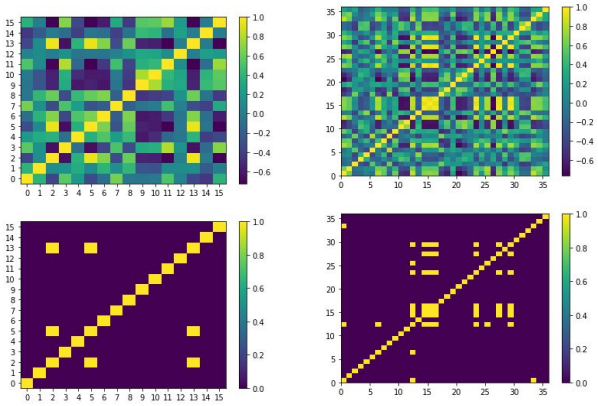


Fig. 5: Correlation matrices

or highly correlated to others. Each prototype is a vector which represents an average member of their cluster. Because some letters can be written in many different ways the network can't learn all of them and so some letters are not very sharp. There is a number of prototypes representing a variation of the same letter such as neuron 21 and 31, both represent a letter E but at a different angle. Overall the network was able to cluster almost all of the letters except for H which seem like got mixed up with A in neurons 3,10 and 11. There is a number of repetitions for letters C and I.

B. Correlation

It is not a surprise that a lot of prototypes are highly correlated. As can be observed from Table II, this is especially true for networks with high number of output units such as one that is used for final results. Most often this happens because the data gets over-represented or because the learning is not precise enough to capture minor differences within existing clusters. Due to results obtained from decay learning run, it is more likely that this occurs because of the second reason as there is a possibility to reduce number of correlated outputs.

To calculate the correlation the training data set is passed trough the network again and the output is recorder. All units that fire over a certain threshold (0.5) are correlated and thus get assigned a value of 1 and all other units -1. Then, the following correlation formula was used to calculate the correlation between two prototypes:

$$Corr_{ij} = \frac{1}{N} \sum_{\mu=0}^{N} p_i^{\mu} p_j^{\mu} \tag{4}$$

This formula produces a final correlation coefficient matrix which is displayed in Fig. 5 at the top left and right for 16 units and 36 units respectively. A further calculation can be undertake assigning all the values above give threshold (0.8 for the purpose of this experiment) to 1 and 0 to the rest. This give a clear count of highly correlated units which is also displayed in the Fig. 5 at the bottom. Studying carefully and comparing the prototypes with the right correlation matrix will confirm that give output indeed look very similar. Those are mostly representing letters I and C.

APPENDIX

A. Reproducing Results

The results for the final experiment can be reproduce by running **python3 comp_learn.py** in the same directory as **letters.csv**. To add or remove different configurations, **line 240** can be edited. Currently only **conscince** and **dynamic_eta** (decay learning) are turned on. To turn on noise, set **noise = True** and to turn on leaky learning set **leaky = True**

B. Code Snippets

```
if dynamic_eta: # if learning rate is set to decay
    eta = eta0*(t**(-eta_alpha))

# get randomly generated index in the input range:
i = math.ceil(m*np.random.rand())-1

x = data[:,i] # pick training instance using the random index

h = W.dot(x)/letters # get ouput firing

# scale values to be between 0 and 1:
h = np.interp(h, (h.min(), h.max()), (0, 1))

k = np.argmax(h) # index of the winning neuron

if conscience: # if conscience algorithm is used
    out = np.zeros(letters)
    out[k] = 1
    prob += 0.001*(out-prob) # adjust the probabilities
    bias = 1*((1/letters)-prob) # calcualate the bias for each neuron
    h -= (1-bias) # subtract the bias from the output
    k = np.argmax(h) # select a new winner

if noise: # if noise is used
    # longtail distribution
    nois = np.random.lognormal(0.01,0.65,letters)/100
    h += nois
    k = np.argmax(h)


counter[k] += 1 # increment counter for winner neuron

# calculate the change in weights for the k-th output neuron:
dw = eta * (x.T - W[k,:])
W[k,:] += dw # weights for k-th output are updated

abs_dw = np.mean(np.abs(dw))
wCount[t] = wCount[t-1] * (((window-1)/window)+abs_dw/window)

if leaky: # if leaky learning is used
    # calculate the change in weights for all other neurons:
    dw = leaky_eta * (x.T - W[not k,:])
    W[not k,:] = W[not k, :] + dw # weights for all other are updated
```

Fig. 6: Learning algorithm + optimisation: The code above is a version of online learning algorithm. Straight away, there is an option to run a decay learning algorithm where the learning rate is adjusted according to the following formula provided in [1] Then, a random input vector is chosen and ran through the network. The output is collected and the maximum is selected representing the winning neuron.

At this point there is an option to adjust the output using conscience algorithm by subtracting biases or by adding noise from a long-tail distribution. If any of those are chosen the new winning neuron is selected. Change in weights are calculated according to the Eq.3. There is also an option to run the leaky learning algorithm where all other weights are also adjusted but by a much smaller proportion.

```
def normalise(arr):

    norm_arr = arr/LA.norm(arr)

    return norm_arr
```

Fig. 7: Normalisation: method takes in an array and returns a normalised version of it. This is done using a linalg numpy module where the vector is divided by its size (l2 norm). After normalisation the square sum of all the elements in the arrays is 1. This is an important part of competitive learning algorithm as without normalised weights and data it will fail.

```
def init_weights(params, data):

    letters = params['letters']

    [n,m] = np.shape(data)

    # Weight matrix (rows = output neurons, cols = input neurons):
    W = np.random.rand(letters,n)

    W = normalise(W)

    return W
```

Fig. 8: Weight Initialisation: the weights are initialised at random. The rows of the weight matrix represent output units, while columns represent input units.

```
def correlation(W, data, params):

    letters = params['letters']

    corr = np.zeros((letters, letters))
    threshold = 0.5

    data = data.T

    for x in data:
        out = W.dot(x) # run the data point trough the network
        # scale values to be between 0 and 1:
        out = np.interp(out, (out.min(), out.max()), (0, 1))

        state = np.zeros(letters)
        state[out <= threshold] = -1
        state[out > threshold] = 1

        corr += np.outer(state, state)

    corr /= data.shape[0]

    return corr
```

Fig. 9: Correlation matrix calculation

```
def output_noise(W, params):

    letters = params['letters']

    out_noise = np.zeros(letters)

    dim = W[0,:].shape[0]
    h = w = int(math.sqrt(dim))

    M = [[1, -2, 1],
        [-2, 4, -2],
        [1, -2, 1]]

    for i in range(0,letters):
        image = W[i].reshape((h,w))
        noise = np.sum(np.sum(np.absolute(convolve2d(image, M))))
        noise *= math.sqrt(0.5 * math.pi) / (6 * (w-2) * (h-2))
        out_noise[i] = noise*1000000

    return out_noise
```

Fig. 10: Image noise calculation: a method described in [2] was used to calculate noise.

REFERENCES

[1] John A Hertz, Anders Krogh, and Richard G Palmer. *Introduction to the theory of neural computation.* 1st ed. Perseus Books, 1999.

[2] J. Immerkr, *Fast Noise Variance Estimation, Computer Vision and Image Understanding*, Vol. 64, No. 2, pp. 300-302, Sep. 1996

[3] Duane DeSieno, *ADDING A CONSCIENCE TO COMPETITIVE LEARNING*, 1988