University of Sheffield

COM3240 - Adaptive Intelligence

Collective Report

Uddhav Agarwal

Department of Computer Science

April 1, 2019

Question 1

This is a simple competitive learning algorithm on a one layer network. In theory, the output neurons are connected to the set of inputs via excitatory connections. The output neuron with the largest net input i.e. the product of the weight and the input vector is the 'winner' neuron. In this algorithm a set of output neurons is generated with random weights. An input vector is chosen at random for which the activation value is calculated. The weight for the 'winner' neuron is updated and for the losers based on several heuristics and tuning methods (described in section 2). The process is repeated for a set number of iterations and the output units are displayed.

This algorithm is online as the weights are adjusted for each training item based on the difference between computed outputs and the training data target outputs. If batch training was implemented then the adjustment are gathered over all training items, to give an average set of values which are then applied to each output neuron.

The training data and the associated weights in this algorithm are normalised. In competitive learning the only one of the output units, called the winner can fire at a time. The winner is normally the unit with the largest net input h. For weight vector $\mathbf{w}$ and input vector $\mathbf{x}$, we have:

$$h = \mathbf{w} \cdot \mathbf{x}$$

Now we know that,

$$\mathbf{w} \cdot \mathbf{x} = |\vec{\mathbf{w}}| \, |\vec{\mathbf{x}}| \, cos\theta$$

Normalising removes geometrical biases towards some of the dimensions of the data vectors and ensures that all the weight vectors and the input unit vectors have the same length and are treated 'fair'. If $\mathbf{w}$ is not normalised then the larger values become dominant and always win (Hertz 2018).

Question 2

There are three optimisation techniques used in this algorithm to improve the accuracy and the overall detection: leaky learning,gradual decay of the learning rate and initialising the weight vector to the input vectors.

In leaky learning (See Listing 4)in addition to the "winning" neuron the weight for the losing neurons is also updated but with a much smaller learning rate. We do this because some weight vectors may lie from the input vectors and as a result it never wins and hence never learn. Updating the losing neurons ensures that it move in the direction of the input neuron clusters.

Another way to prevent winner-takes-all situation is that the learning rate can be gradually lowered or decayed (See Listing 3) as proposed by Fritzke (1997). Exponential decay in combination with harmonic decay has been used in this function (see Listing 2). As the algorithm reaches a minima, a lower learning rate will settles at a lower point than it might have for the constant learning rate.

Initialising the weight vector to the input units (See Listing 1) helps prevent dead units as the vectors are usually placed in the general vicinity of the inputs.

The configuration was run for various initial conditions such as where all optimisation techniques were used, none were used and etcetera. The experiment was ran 5 times in each configuration for 16 output neurons. As the number of units each process remained more than zero, any neuron that fires <4% is counted as dead. Table 1 shows the mean and standard deviation of the number of dead neurons.

From the results produces it can be observed that learning rate decay and initialising the output neurons to the sample works best but may lead to outputs that look similar. Leaky learning did not work very well is the test cases as the output were capturing the same information and could not be distinguished.

Question 3

Using all the optimisation methods and 16 output neurons, the weight change over time is shown in Figure 1. The x-axis and y-axis are on a log scale. As online learning was employed for this algorithm, the graph has been smoothed with moving average. The graph clearly shows that at around the 105 iteration the moving average becomes minimum at which stage the network has finished learning. Any further iterations are redundant as adjusting the

	NONE	ALL	LK+D	LK+W	D+W	LK (rate: 10^{-6})	LK (rate: 10^{-6})	W
Mean	10.33	7.00	10.67	8.00	6.33	9.67	9.33	8.33
Standard Deviation	0.58	1.00	0.58	1.00	0.58	0.58	0.58	1.53

Table 1: Table showing the results for experiments under different configurations. KEY: NONE = no optimisations applied, ALL = all three optimisations applied, LK = Leaky Learning, D = Learning Rate Decay, W = using the inputs to initialise weights

output neuron will not have much effect to the other nearby inputs.

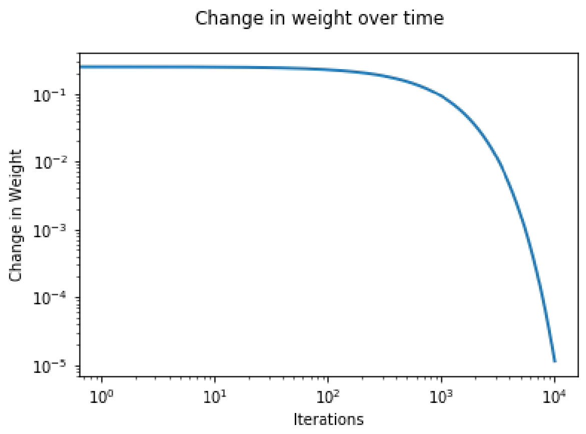

Figure 1: Plot showing the average change in weight over time of the output neurons

Question 4

With all optimisation techniques applied, the network found 16 prototypes as seen in Figure 2. Each prototype represents average 'shape' or the alphabet of their cluster. The more brighter 'shape' inside a prototype represents the input for which the output fires for the most. It is interesting to note that some of the prototypes show the same letter for example the 1st and 2nd prototype seem to be a 'J'. This is because the two prototype are similar but have a non-identical contrast drawn pattern. The programs lack of ability to distinguish between variations may also be the reason why some of the prototypes look like a combination of two shapes for example the 9th prototype look like a 'B' and a 'D'.The similarity between different prototypes can be deduced by finding the correlation matrix. Similar letters fire together and hence have a higher correlation as shown in Figure 3.

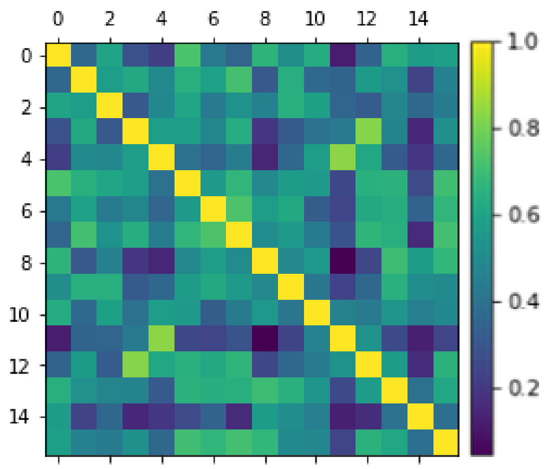

Figure 3: Correlation matrix for the prototypes displayed in Figure 2.

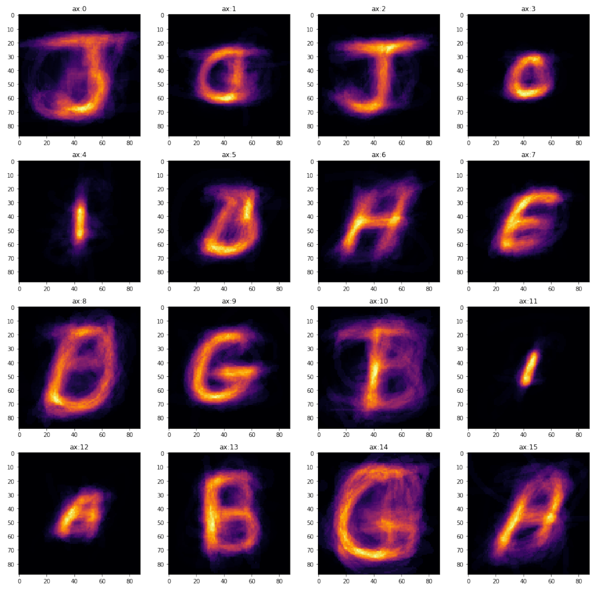

Figure 2: Figure showing the 16 prototypes detected by the network

Appendix A: Reproducing results

All the results (Table 1), prototypes (Figure 2), correlation matrix (Figure 3) and change in weight (Figure 1) can be produced by running the `results` function in `competitive_learning.py`. The function `results` takes 8 arguments:

- train: array

  The normalised training data

- eta: int, optional

  The value of the learning rate for the winner neuron. Default is set to 0.06

- eta_l: int, optional

  The value of the learning rate for the loser neuron in case of leaky learning. Default is set to 0.000006

- eta_d: int, optional

  The lower limit for the learning rate in case of gradual learning rate decay. Default is set to 0.03

- digits: int, optional

  The number of output neurons or prototypes. Default is set to 16.

- weight: ['random', 'input'], optional

  Choose whether to initialise the weights randomly or to the input weights. Default is set to random

- leaky: ['not_leaky', 'leaky'], optional

  Choose whether to employ leaky learning. Default is set to not_leaky.

- decay: ['decay'], optional

  Choose whether to decay the learning rate. By default no decay

Additionally, the `train` and its dimensions `n` and `m` are extracted by using the `getNormInput` which requires one argument which is the URL for the input file from which the data is to be extracted from. The default file used is `'letters.csv'`.

The plots and the measurement of improvement (mean and standard deviation) are saved in the root directory as images and a text file respectively.

Appendix B: Code

This section contains some code snippets that are integral to the whole algorithm. NOTE: The code displayed here has been modified for understanding the program. Please refer to the actual code file for the full running code.

Initialising the Output Neurons

```
 # Random weight matrix #
if (weight == 'random'):
    W = winit * np.random.rand(digits,n
    )

# Random weight matrix #
elif (weight == 'input'):
    W = np.zeros((digits,n))
     # randomly assigning input weights
     to output
     for t in range(digits):
         i = math.ceil(m * np.random.
     rand())-1
         y = n_train[:,i]
        W[t,:] = y
```

Listing 1: Initialises the data either randomly or to the input weight based on the user input.

Normalising the Output Neurons

```
# Normalising the weight vector
normW = np.sqrt(np.diag(W.dot(W.T)))
normW = normW.reshape(digits,-1)

W = W / normW
```

Listing 2: Normalising the output weights to prevent an 'unfair' competition

Gradual Learning Rate Decay

```
eta = (eta * ((eta_d/eta) ** (t / (tmax
    ))) - 1/tmax)
```

Listing 3: One-line function do decrease the learning rate. Here eta and `eta_l` are the initial and the final learning rate values. The value of t ranges from 1 to tmax.

Leaky Learning

```
# only update the winner
if (leaky == 'not_leaky'):
    dw = eta * (x.T - W[k,:])
    W[k,:] = W[k,:] + dw
# leaky learning loop
elif (leaky == 'leaky'):
    for q in range(digits):
        if (q == k):
            dw = eta * (x.T - W[q,:])
            W[q,:] = W[q,:] + dw
        if (q != k):
            dw = eta_l * (x.T - W[q,:])
            W[q,:] = W[q,:] + dw
```

Listing 4: Leaky learning code. Here eta is the learning rate for winning neuron and `eta_l` is the learning rate for the losing neurons

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

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

    # get output firing
    h = W.dot(x)/digits
    h = h.reshape(h.shape[0],-1)

    # get the max and its index in the
    output firing vector
    output = np.max(h)
    k = np.argmax(h)

    # increment counter for winner
    neuron
    counter[0,k] += 1

    # calculate the change in weights
    for the 'winner'
    dw = eta * (x.T - W[k,:])

    # get closer to the input
   W[k,:] = W[k,:] + dw

    # % weight change over time (
    running avg)
    wCount[0,t] = wCount[0,t-1] * (
    alpha + dw.dot(dw.T)*(1-alpha))
```

Listing 5: This is the online implementation of the competitive learning algorithm

Learning Algorithm

References

Fritzke, Bernd (1997). "Some competitive learning methods". In: *Artificial Intelligence Institute, Dresden University of Technology.*

Hertz, John A (2018). *Introduction to the theory of neural computation.* CRC Press.