Machine Learning - Assignment 1

Francesco Saverio Zuppichini

October 12, 2017

1 The Perceptron

1.1 Question 1

1.1.1 Vectorized equations

The vectorized equation for a single perceptron

$$output = XW + b \tag{1}$$

Where $X = x_1, ...x_n$, $W = w_1, ..., w_n$. We denote y the output of the perceptron

1.1.2 Mean Squared Error

The mean squared error function for our single perceptron

$$E(w) = \frac{1}{N}\sum_{i=1}^{N}(\underbrace{y(x_i, w_i)}_{\text{predicted}} - \underbrace{t_i}_{\text{actual}})^2) \tag{2}$$

One trick that is usual done is to multiply equation (2) by $\frac{1}{2}$, so when we take the derivative the 2 goes away. This is called One Half Mean Squared Error.

1.1.3 Derivate of the error with respect to the weights

In order to reduce our error function and adjust each weight we need to compute the first derivati with respect to the weights. We will use the modified version of equation (2) called One Half Mean Squared Error.

$$\frac{\delta E}{\delta w_i} = y - t \tag{3}$$

1.1.4 Gradient Descend

The gradient descend is an iterative optimisation algorithm that follow the direction of the negative descent in order to find the minimum of an objective function. It is can be used as Learning Algorithm since it allows to reduce our error function, equation (2), and adjust the weights properly. It's equation

$$w_{k+1} = w_k - \eta \nabla E(w_k) \tag{4}$$

Where η is the step size, also called **learning rate** in Machine Learning. This parameter influence the behavior of Grandient Descent, a small number can lead to local minimum, while a bigger learning rate could "over-shoot" and decreaasing the converge ration.

For this reasons, numerous improvements have been proposed to avoid local minima and increase its convergence ration. Some of them are: Conjugate Gradient and Momentum.

1.2 Implement the MSE and dMSE

You can find them in *MSE.py*

1.3 Implement the function forward and backward

For the *forward* and *backward* function you can find the code in *Perceptron.py*. You can find *train_one_step*in *sketron.py*.

1.4 Implement the *run_part1*

You can find the implementation in the code. The plot showed in figure 1 the final result after 15 steps using a learning rate of 0.02

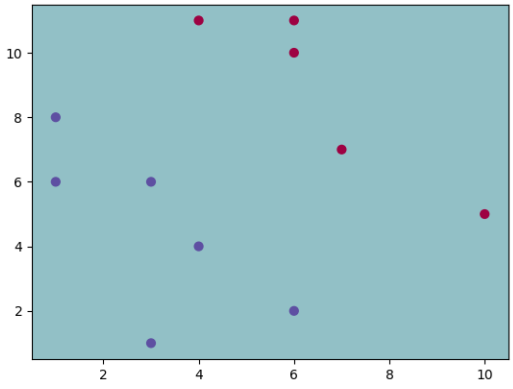

Figure 1: *run_part1* plot

2 A Neural Network

2.1 Implement the activation functions

You can find them inside *activation.py*

2.2 Question 2

2.2.1 Forward pass

In order to calculate the forward pass of a Neural Network we need to compute the activation of each layer, l, ad use it as input of the next one until we reach the output layer.

Equation 5 shows the activation a of layer l for the j-th neuron on that layer.

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \tag{5}$$

Where w_{jk}^l is the connection from neuron k in the $l-1$ layer to j, a^{l-1} is the activation of the previous layer and b_k^l is the bias of k-th neuron in the l layer. With this in mind, we can rewrite 5 in a efficient vectorized form

$$a^l = \sigma(W^l a^{l-1} + b^l) \tag{6}$$

2.2.2 delta rules

A Neural Network try to change its weights in order to decrease the error function, E. We define δ_j^l the output error of neuron j in layer l

$$\delta_j^l = \frac{\delta E}{\delta z_j^l} \tag{7}$$

Stringly speaking, δ_j^l, is how much the error function changes by changing the weighted input on that layer. Applying the chain rule, equation 7 becomes

$$\delta_j^l = \frac{\delta E}{\delta a_j^l} \frac{\delta a_j^l}{\delta z_j^l} \tag{8}$$

Knowning that $a_j^l = \sigma(z_j^l)$ equation 8 can be rewritted

$$\delta_j^l = \frac{\delta E}{\delta a_j^l} \sigma'(z_j^l) \tag{9}$$

2.2.3 Derivatives of the weights

We want to compute $\frac{\delta E}{\delta w_{jk}^l}$. Applying the delta rule

$$\frac{\delta E}{\delta w_{jk}^l} = \frac{\delta E}{\delta z_j^l} \frac{\delta z_j^l}{\delta w_{jk}^l} = \frac{\delta E}{\delta a_j^l} \frac{\delta a_j^l}{\delta z_j^l} \frac{\delta z_j^l}{\delta w_{jk}^l} \tag{10}$$

$$\frac{\delta E}{\delta w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{11}$$

2.3 Implement the functions forward and backward of the Neural Network class.

2.4 Train Network

2.4.1 Split the data into a train set and a test set

I decide to split my Training set and Test set by a ratio of 80:20. If we choose a smaller train set we may under-train the network, on the other hand we if we shrink the test set we may *overfit*.

2.4.2 Initialize the weights randomly and find a good learning rate to train the network until convergence

I initialialized the weight using following formula:

$$w = random(n)/\sqrt{2}$$

Where n is the size of the input. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. [1]

Using *numpy*:

```
np.random.randn(in\_size,out\_size)/np.sqrt(in\_size)
```

2.4.3 Plot the learning rate of at least 5 different learning rates on both sets

TODO

2.4.4 Plot the boundary of the model which converged the most on the training data

TODO

2.5 Explain if this is a good model for the data

3 Further Improvements

3.1 Momentum

[1] http://cs231n.github.io/neural-networks-2/