Q1 – Perceptron

Using one perceptron with two inputs, we first initialize the input weights and the bias to 0. It could also be randomly initialized but it doesn't make much of a difference. Afterwards, using stochastic gradient descent, the weights are trained. First, the data and the weights are linearly calculated (activation) and according to its value, the error is defined. We then use the error and learning rate to update the weights and biases in a few iterations.

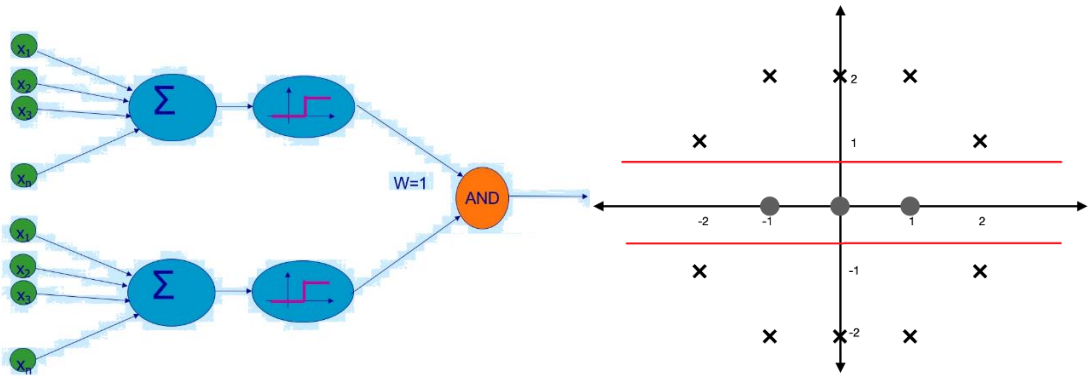
Q2 – Binary Classification

The only difference between this question and the previous one is in the data and the plotting. Epoch and learning rate values have been obtained in trial and error. The data preparation is straightforward.

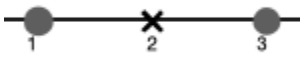
Q3 – Madaline

Madalines consist of multiple Adalines, so instead of one output, we have multiple outputs and weight matrices. At the end, the linear classifications are ANDed with eachother so it gives us the opportunity to create convex areas with the lines.

Since Madaline can be used to classify data in convex polygons, image 1 can be classified to its appropriate classes using two Adalines. Since there is only one convex region, no hidden layers are required.



But in image two, because of the situation shown below, there is no convex shape that can separate those two classes:



Q4 – MLP

MLPs don't understand more than one dimension. For this reason, we must resize our data to simple vectors. Each MNIST data is a 28x28 picture, so we have 784 pixels. The MNIST targets, which are just numbers (and numbers can take any value), are not categorical. With to_categorical, we can turn the numbers into categorical data. For example, if we have a trinary classification problem with the possible

classes being {0,1,2}, the numbers 0, 1 or 2 are encoded into categorical vectors [source]. If the correct class for the image is 1, the categorical vector looks like this: [0, 1, 0] which is self-explanatory. We have prepared our data.
For the MLP itself, I have used 3 layers with ReLU as their activation function and one with softmax. Before these, the visible layer takes all 784 pixels and they incrementally get into 10 classes in the end. Categorical cross-entropy is used as our loss function and the optimizer is Adam in its default format available in the Keras library. The model is fitted to the training data and uses the test data for validation as the question required. After a few runs, I realized 20 iterations (epochs) is accurate enough and doesn't cause any overfitting. The loss and accuracy plots are drawn with standard matplotlib functions.

Q5 – Back-Propagation

This neural network has 3 layers: an input with 784 neurons, a hidden with 10 neurons, and an output with (naturally) 10 neurons. There are three parts to training the neural network. Forwarding, back-propagation and updating. At the very first step, the weights and biases are randomly generated and we work on improving them for each iteration.

Forwarding

The first layer takes the pixels as inputs (A0). Let's say Z1 is the unactivated first layer and the first weights and biases are W1 and B1, respectively. The equation is as follows:

$$Z1 = W1.A0 + B1$$

Then of course we need to activate the layer, which is done with the ReLU function. Let A1 be the activated first layer:

$$A1 = ReLU(Z1)$$

But for the last layer, I'm using the softmax activation function because we want the outputs to be probabilities and give us the appropriate result. Which results in:

$$Z2 = W2.A1 + B0$$

$$A2 = Softmax(Z2)$$

Back-Propagartion

But, of course, forwarding isn't enough. We need good weights and biases. For this, we'll go the opposite way to figure out how we can change our variables to get the desired result. First, we need a one-hot encoded version of the output, hence the function implemented for it. If we declare a loss function and minimize it, the accuracy of the model improves. We do so by subtracting the derivative of the loss function for each parameter over many rounds of gradient descent. If J is the loss function and alpha is the learning rate we have:

$$W1 := W1 - \alpha \frac{\delta J}{\delta \mathrm{W1}}$$

$$B1 := B1 - \alpha \frac{\delta J}{\delta \mathrm{B1}}$$

$$W2 := W2 - \alpha \frac{\delta J}{\delta \mathrm{W2}}$$

$$B2 \coloneqq B2 - \alpha \frac{\delta J}{\delta \mathrm{B2}}$$

The loss function I chose is $\sum_{i=0}^{c} 1\, y_i \log(\hat{y}_i)$ and we're looking for its derivatives. For simplicity I'll write them as dW1, dB1, dW2 and dB2. For this, we'll start by calculating dA2 (going backwards through our neural network) which is simply $\mathrm{dA2} = \mathrm{Y} - \mathrm{A2}$. The rest of the calculations is as follows (with m being the size of the dataset and g being the activation function which in this case is ReLU):

$$d\mathrm{W2} = \frac{1}{m} d\mathrm{Z2} A1^T$$

$$d\mathrm{B2} = \frac{1}{m} \Sigma d\mathrm{Z2}$$

$$d\mathrm{Z1} = (W2)^T d\mathrm{Z2} .* g1'(\mathrm{Z1})$$

$$d\mathrm{W1} = \frac{1}{m} d\mathrm{Z1} X^T$$

$$d\mathrm{B1} = \frac{1}{m} \Sigma d\mathrm{Z1}$$

Updating

After having all the derivates needed, we can update the parameters which for each parameter P, is done like this:

$$\mathrm{P} \coloneqq \mathrm{P} - \alpha d\mathrm{P}$$

We'll do this process a number of times until we are satisfied with the accuracy it results in. With 0.1 as the learning rate and 500 number of iterations, the accuracy on training data has been 0.85 in average and it's a little less on the test data (around 0.83 in average).

Links Used

http://www.cs.bc.edu/~alvarez/ML/gradientSearch.pdf

https://www.machinecurve.com/index.php/2019/07/27/how-to-create-a-basic-mlp-classifier-with-the-keras-sequential-api/

https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/