Laboratory 2 – Feedforward Network and Backpropagation

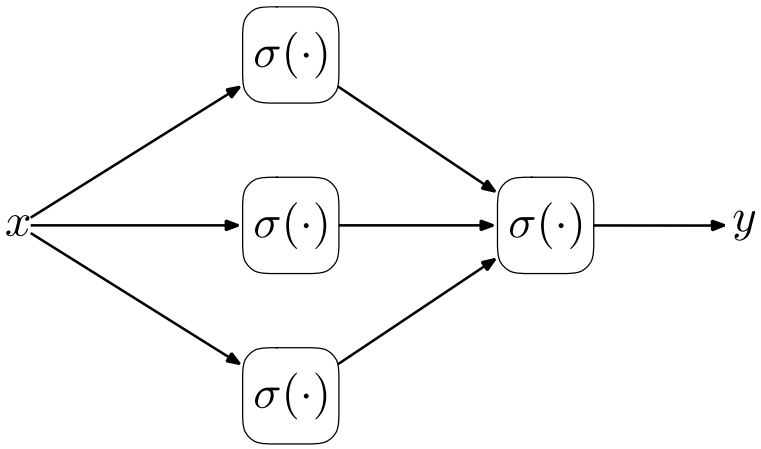
Objectives of this lab:

1. *Understand Backpropagation*
2. *Write a neural network with one or more hidden layers*
3. *Solve the XOR*
4. *Understand how to build general classifiers*

During the previous lab we noticed that a single neuron is not able to solve a non-linearly separable problem like the XOR. In this class we will expand the capabilities of our initial network, will move beyond a single neuron and we will solve arbitrarily complex datasets.

To start, let's open again the file that we developed in the previous laboratory (save this as a new file, as we will heavily modify the code).

We want to create a network with the following structure:



$$s = w^T x + b$$

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

To do so, let's modify the way we initialize our weights and biases

```
16
17 W1 = np.random.randn(3,2)
18 B1 = np.random.randn(3)
19 W2 = np.random.randn(1,3)
20 B2 = np.random.randn(1)
21
```

Can you explain why we set the dimensions of the weights as (3,2) and (1,3)?

After that let's add a forward function to reflect the network topology that we want to replicate:

```
22
23 def sigm(X, W, B):
24
25     M = 1/(1+np.exp(-(X.dot(W.T)+B)))
26
27     return M
28
29 def Forward(X, W1, B1, W2, B2):
30     #first layer
31
32     H = sigm(X,W1,B1)
33
34     #second layer
35
36     Y = sigm(H,W2,B2)
37
38     # We return both the final output and the output from the hidden layer
39
40     return Y, H
41
```

Of course, due to the presence of the hidden layer, we need to use the backpropagation algorithm to update the weights.

$$E = (z - y)^2$$

$$w_i' = w_i + \eta \frac{dE}{dw_i}$$

$$b' = b + \eta \frac{dE}{db}$$

It is very helpful to try and derive the expressions for the update of the weights yourself, so before moving on, try and find an analytical expression for them. Remember that the derivative of the sigmoid function and the chain rule (in general) can be expressed as

$$\frac{\sigma(s)}{ds} = \sigma(s)(1 - \sigma(s))$$

$$\frac{dy}{dx} = \frac{dy}{ds}\frac{ds}{dx}$$

Note: the constant 2 that appears in the derivative of the error can be omitted in the code as that's included in the learning rate.

Now we can write the code for the updates of the various parameters:

```
39
40 def diff_B2(Z,Y):
41     dB = (Z-Y)*Y*(1-Y)
42     return dB.sum(axis=0)
43
44 def diff_W2(H, Z, Y):
45     dW = (Z-Y)*Y*(1-Y)
46     return H.T.dot(dW)
47
48 def diff_W1(X,H,Z,Y,W2):
49     dZ = (Z-Y).dot(W2)*Y*(1-Y)*H*(1-H)
50     return X.T.dot(dZ)
51
52 def diff_B1(Z,Y, W2,H):
53     return ((Z-Y).dot(W2)*Y*(1-Y)*H*(1-H)).sum(axis=0)
54
```

Notice that, unlike the previous lab, we are not making use of the sigmoid function inside the update rules, instead we feed them the outputs from the middle layer (H, in this specific example). The results are the same and which expression you use is simply a matter of readability and compactness of the code.

```
60 learning_rate = 1e-2
61
62 for epoch in range(10000):
63
64     Y, H = Forward(X, W1, B1, W2, B2)
65
66     W2 += learning_rate * diff_W2(H, Z, Y).T
67     B2 += learning_rate*diff_B2(Z, Y)
68     W1 += learning_rate * diff_W1(X, H, Z, Y, W2).T
69     B1 += learning_rate * diff_B1(Z, Y, W2, H)
70     if not epoch % 50:
71         Accuracy = 1 - np.mean((Z - Y)**2)
72         print('Epoch: ', epoch, ' Accuracy: ', Accuracy)
73
```

Once that is done, just update the training process and check the results:

Congratulations! You wrote your first functioning deep network. Every other one that you will create, from now on, will be based on the same basic principles that you just learnt.

Now, let's take a look at a different activation and error function, normally used for classifiers: the softmax and the cross-entropy function.

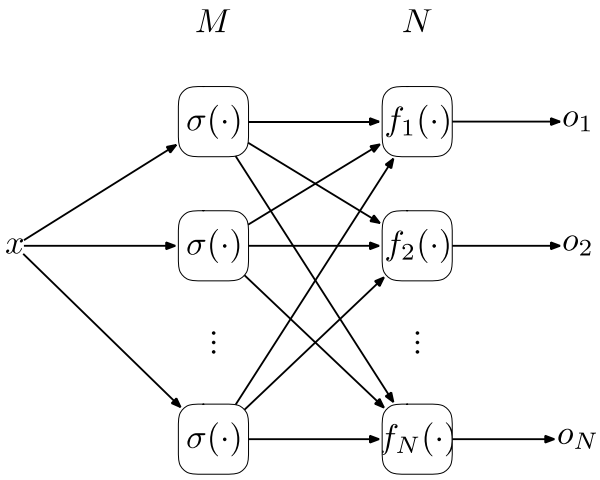
Suppose your training data was made by two different sets:

- A) 2 Points sampled according to a uniform distribution in the range [0,5];
- B) 2 Points sampled according to a uniform distribution in the range [0,10].

You need to create a network that, given two random points in the interval [0,10], gives as an output the probability of belonging to class A and to class B.

You'll notice immediately that the following problem does not have a clear-cut answer as the ones we solved before: for example, something like [3.5, 4.2] could, in theory, belong to both classes (but it belongs to one of the two classes with higher probability!). The same thing happens in reality: most often we are not able to distinguish exactly between a number N of classes and what we need is a "best guess". Can you name an example that could be represented by this situation?

The classical way of approaching a classification problem, is to make use of the following architecture



Where M is the number of neurons in the hidden layer and N is the number of classes to classify (2, in this exercise). The activation function for the output layer is the softmax function:

$$o_i = f_i(s) = \frac{e^{s_i}}{\sum_{j=1}^{N} e^{s_j}}$$

This function rescales your outputs in the range [0,1] and sets their sum equal to 1 so that the output represents the "probability" that the corresponding input would belong to one of the classes. For instance, a [0.5, 0.5] output would mean that there is an equal chance that your input belongs to class A or to class B. This representation is ideal when your label z, is a "one-hot vector", where there is 1 one (the correct class) and N-1 zeros (the incorrect ones). For instance, imagine that you were trying to teach a network to distinguish between images of cats and dogs. The correct labels then would be:

$$z_{\text{dog}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad\qquad z_{\text{cat}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Moreover, for this kind of networks, it is customary to make use of the cross-entropy loss as an error function:

$$H = -\sum_{j=1}^{N} z_j \log(o_j)$$

Where z_j represents the j-th entry of z. Cross-entropy loss, measures the performance of a classification model whose output is a probability value between 0 and 1, and the error increases as the predicted probability deviates from the actual label. A perfect model would have a loss of 0. For instance, say that in this exercise with two classes, you had an input that certainly belongs to the second class (e.g, [10,10]) so that its corresponding output would be [0,1]. If the output of your network was something like [0.5, 0.5] your error would be:

$$H = -1\log(0.5) - 0\log(0.5) = -\log(0.5) \approx 0.693,$$

Whereas if your output was (0.05, 0.95), which is much closer to the correct one, the error would be:

$$H = -1\log(0.95) - 0\log(0.05) = -\log(0.95) \approx 0.051.$$

The softmax function and the cross-entropy loss match perfectly one another and it can be shown that

$$\frac{dH}{ds_i} = o_i - z_i$$

Which makes the expression for the backpropagation, very simple to write. Accordingly, try to modify the code that we wrote for the XOR function to solve the previous classification problem (the solution is provided in a separate file).

Note: due to the random initialization you might have different results. Try to run it a few times and see what happens. Try also to modify the number of neurons and see the results.