Homework 02

IANNWTF 20/21

Submission until 7th Nov 23:59 via https://forms.gle/8qwiuUXCfLjKGeg67

This week we will implement a Multi-layer Perceptron and train it to solve logical gates and the XOR problem. To do so, we will implement backpropagation ourselves.

We know this is a bit tedious and yes, it is true that you will never have to do it from scratch again. However, we think it is the best way to really understand what is going on and appreciate the beauty of having libraries like tensorflow.

You can choose to do this homework in a Jupyter notebook or using different scripts (one module for each class). We will not use any tensorflow / keras libraries yet. For this homework, you will only need numpy. Therefore, make sure to `import numpy as np` in the beginning.

We will give you a step-by-step guide to help you get started. However, feel free to also try it yourself and only come back to the instructions when you don't know what to do anymore. Some parts also include hints (denoted by the superscript numbers) - you can find them at the end of this document. Do not read them immediately, but think about it first, and only go to them if you're stuck.

1 Preparation

For this homework you will use sigmoid as an activation function. Think about the following questions (you do not have to hand in the answers, they are just for your own recap)

- What is the purpose of an activation function in a NN in general?
- What's the advantage of e.g. sigmoid over the step function (threshold function)?
- How does sigmoid look like (the formula as well as the graph)?
- What is the derivative of sigmoid?

Implement a function `sigmoid(x)` and a function `sigmoidprime(x) (the derivative`.

2 Data Set

The training data set will consist of possible **inputs** and their corresponding **labels**. We are training the network on logical gates (and, or, not and, not or, xor == exclusive or). We will create the inputs and labels ourselves.

What are possible **inputs** to the logical gates? [1]

For each of the logical gates you will need an array of **labels** (= the true solution that the network is supposed to output), one corresponding to each input pair. [2]

3 Perceptron

Our multilayer-Perceptron will consist of single Perceptrons. So we will need a class `Perceptron`.

Think about what a Perceptron consists of. [3]

When you create a Perceptron, it should receive an integer argument called `input_units` specifying how many weights are coming in to your Perceptron. In the beginning, random values should be assigned to the weights and the bias. [4] Also assign the learning rate `alpha = 1`. [5]

The Perceptron should have a function `forward_step(self, inputs)` that calculates the activation of the perceptron. Use sigmoid as activation function.

Then you'll need a function `update(self, delta)` which updates the parameters. To do so, compute the gradients for weights and bias from the error term δ. (This error term will be passed to the function when the backpropagation of the parent class `MLP()` is called - see next section.
Compute the gradients using:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \; {a_j}^{(l-1)}$$

And then update the parameters using:

$$\theta_{new} = \theta_{old} - \alpha \nabla L_\theta$$

[6]

4 Multi-Layer Perceptron

Further, we will need a class `MLP()` that can perform a forward and backprop-step.

Initialize the MLP with **1 hidden layer** that has **4 Perceptrons**. Initialize **1 output neuron**. [7] Define the following functions of the class:

- In the `forward_step` the inputs are passed through the network. [8]
- In the `backprop_step` the parameters of the network are updated. [9]

5 Training

As a loss function for training, we will use the squared error $(t - y)^2$. This loss is the sigmoid output vs. the target (=label in dataset). But as discussed in the lecture, we want to introduce an additional measurement of the performance of the network: This is the accuracy measure. While the loss compares the distance of our network to the ground truth, the accuracy makes a less qualitative statement about the performance of our network, but quantitatively tells us how our network is doing. To do so, we introduce a threshold, in our case we use 0.5 and define that if the network outputs a value bigger than 0.5 and the target is 1, it counts as a correct classification. If target is 0 a correct classification will be a value smaller than 0.5 respectively. The accuaracy is then defined as the ratio of correct classification vs total classifications performed.

The training procedure thus should consist of the following steps:

1. Create an instance of an MLP class.
2. Train the instance for 1000 epochs.
    a) In each epoch, loop over each point in your dataset once.
        i. For each data point, perform a forward and a backward step.
        ii. Record the accuracy and the loss for each point.

For the purpose of visualization and also to monitor the performance of your network, you should keep track of the epochs and the average loss and accuracy for each epoch.

6 Visualization

Visualize the training progress using matplotlib. Create one graph with the epochs on the x-axis and the average loss per epoch on the y-axis. Do the same for the average accuracy per epoch.
If your MLP trained correctly, the loss should come down to zero and the accuracy should go up to 1 in most cases. Due to random weight initialization the accuracy might not reach 1 sometimes. In that case just rerun the MLP initialization and the training.

Notes

[1] The logical gates take as input two binary digits (either 1 or 0), with all possible combinations there should be 4 input pairs. Put them in a 2D numpy array. (The shape of the array should be (4,2))

[2]You will need 5 arrays for each type of gate containing 4 binary digits (0 or 1 corresponding to the input).

[3]weights and a bias

[4]you can use `np.random.randn()` for that

[5]All of this happens in the __init__ function. Make sure you define weights, bias, alpha with `self.` in the beginning so they can be accessed in all functions of the class.

[6]θ denotes the parameters, meaning the weights and biases, α denotes the learning rate, which is 1 in our example, ∇L is the gradient of the error loss of the respective parameter

[7]It might make sense to also initialize a variable `self.output` to store the output.

[8] First compute the activations for the hidden layer. (You might need to reshape the resulting array to feed it to the output neuron. Check `np.reshape(arr, newshape=(-1))`.)
Then feed the activations of the hidden layer into the output layer. Store it in `self.output`.

[9] That means, update the weights and the biases of the output neuron (first) and neurons in the hidden layers (afterwards). For that, first compute the error term δ using this formula:

$$\delta_i^{(l)} = \{ -(t_i - y_i)\ \sigma'({d_i}^{(N)})\ if\ l = N,\ \left(\sum_{k=1}^{m} \delta_k^{(l+1)} w_{ki}^{(+1)}\right) \sigma'({d_i}^{(l)}) \quad else.$$

Then call the `update(self, delta)` function of the respective neuron and hand the delta over.
(This is just a suggestion, there might be different solutions.)