Homework 02

IANNWTF 20/21

Submission until 11th Nov 23:59 via https://forms.gle/H31ckx5251Qg3Ndm6

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. In the Flipped Classroom Session on Thursday we will prepare a Perceptron together. You can use this code and build your MLP on top of that.

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 until now, only numpy. Therefore, make sure to `import numpy as np` in the beginning.

We will give you a step-by-step guide in the following. 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) at the end of the 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 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)`.

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`. You can use the one that we have implemented together in the Flipped Classroom Session (should also be uploaded) or create your own.

If you want to implement it yourself, think about what a Perceptron consists of. [3] When you create a Perceptron, it should receive an int `input_units` with how many weights are coming in to your Perceptron. It should also randomly assign weights and the bias. [4] Also assign the learning rate `alpha = 1`. [5]

np.randn()

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. (It was handed over when the function was called in the `backprop_step` function of the class `MLP()`.) 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]

In the `forward_step` process the inputs through the network. [8]

In the `backprop_step` update the parameters of the network. [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. But as discussed in the lecture, we want to introduce an additinal measurement of the performance of the network: This is the accuracy measure. While the loss compares our pro As a threshold use 0.5. Meaning 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.

Create an instance of the MLP class and train it for a 1000 epochs. One epoch is looping over each point in your dataset once. Perform a forward and backward step for each point in the dataset and record the loss and accuracy. For visualization 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 initialisation the accuracy might not reach 1 sometimes. In that case just rerun the MLP initialisation 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 each containing 4 binary digits (0 or 1).

[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, so 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:

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

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.)