

### 3. Assignment in “Machine Learning for Natural Language Processing”

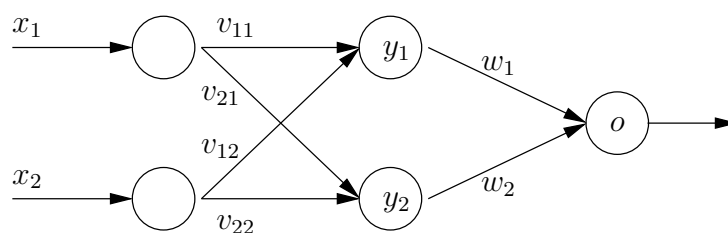
Summer Term 2024

## 1 General Questions

1. What are the main components needed to train a neural network in PyTorch? Explain what each of these components is good for.
2. Suppose we build a neural network that is able to generate text. We need to find a loss function that is used to optimise the model. A friend suggests: “From the lecture, we know that a loss function must give a number that indicates how bad the model is, such that we can minimize this during gradient descent. Why don’t we ask people to rate the quality of generated texts and use them as the loss function?” Assuming we want to optimize our model via backpropagation, what is the problem of this approach?

## 2 Neural Networks and Circuit Diagrams

Given the following neural network with `sigmoid` as activation function:



1. Represent the network as three equations (one each for  $y_1$ ,  $y_2$  and  $o$ )! We do not use any bias in this network.
2. Convert the network to a circuit diagram as the one shown in the lecture!  
*Hint: Draw the diagram as large as possible. You will need space for the back-propagation.*

3. To measure the quality of our weights, we will use the following loss function:

$$L = \frac{1}{2} \cdot (t - o)^2,$$

where  $o$  is the output of the network and  $t$  is the target (correct) label.

Add the loss function to the circuit diagram from the last subtask!

4. Now that we have a circuit diagram, we can rather easily do backpropagation. Determine the partial derivatives for all *weights* ( $v_{11}, v_{12}, v_{21}, v_{22}, w_1$  and  $w_2$ ) in the network!

Assume the following initial values:  $v = \begin{pmatrix} 0.5 & 0.75 \\ 0.25 & 0.25 \end{pmatrix}$ ,  $w = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ .

The inputs are  $(x_1, x_2) = (1, 1)$  and the target label is  $t = 0$ .

Remember the intuitive properties of  $+$  and  $*$  from the slides:

- $+$  just passes the gradient through, both inputs have the same partial derivative as the output
- $*$  “switches” the input, that is: if the gradient after the  $*$  is 2 and the inputs are 1 and 3, then first input gets the partial derivative  $3 \cdot 2 = 6$  and the second input gets  $1 \cdot 2 = 2$ .

### ? Something to think about

5. Backpropagation would enable us to compute a gradient for the *input values*  $(x_1, x_2)$ , too. What could we do with the information about the input’s gradient?

## 3 Python

### Implementing Neural Networks Part 1 — The Forward Pass

In this assignment, you will implement a neural network “library” yourself, using `Python` and `Numpy` (`import numpy as np`). The tool is inspired by PyTorch’s implementation. This week, you will implement the forward pass. Next week, the backward pass and backpropagation will follow.

**Modules** We will follow PyTorch’s code structure when building single parts of our library, such as neural network layers, loss functions, or optimizers. Parts are called

“Modules” in PyTorch, so we will use this terminology. Each of the following modules should have the same structure: A module is a Python class with a `forward` and a `backward` function. The `forward` function calculates the results for the module based on the previous results. The `backward` function calculates the gradients of the module. Today, you only have to code the `forward` functions of the following modules.

**Sigmoid** Implement the `forward` function that takes as parameter a Numpy array of numbers and applies an element-wise sigmoid on this array!

```
class Sigmoid:
    def __init__(self):
        #...
        pass

    def forward(self, x: np.array) -> np.array:
        #...
        pass

    def backward(self, x: np.array, grad: np.array =
        ↪ np.array([[1]])) -> np.array:
        # don't mind me this week
        pass
```

**Mean Squared Error** Implement the `forward` function that takes as parameter two Numpy arrays of numbers and returns the mean squared error between these arrays! The mean squared error is defined as follows:

$$se = \frac{1}{2}(t - o)^2,$$

$$mse = \frac{1}{n} \sum_{i=1}^n se_i,$$

where  $n$  is the size of the vectors,  $o$  is the predicted output of the neural network and  $t$  is the true label.

```
class MeanSquaredError:
    def __init__(self):
        #...
        pass
```

```

def forward(self, y_pred: np.array, y_true:
    ↪ np.array) -> float:
    #...
    pass

def backward(self, y_pred: np.array, y_true:
    ↪ np.array, grad: np.array = np.array([[1]])) ->
    ↪ np.array:
    # don't mind me this week
    pass

```

**Fully Connected Layer** A neural network layer is also a module. If we define one layer as a module, we can create multiple instances of it and send the output of one layer into another layer. This way, we are very flexible in terms of how the data flows through our program. To be very flexible when it comes to the layer input and output sizes, we specify two parameters in this module.

Implement the following class such that we can create a fully connected layer with any input or output size (larger than zero). `self.weights` and `self.bias` are attributes of this class. They store the weights and bias matrix, respectively. For initialization, the weights should be standard-normally distributed. The initial bias values are zero. The `forward` function gets the input and calculates the output of the layer. The input is a 2d array, where each **row** is a data point. This is the same as PyTorch handles data points.

```

class FullyConnectedLayer:
def __init__(self, input_size: int, output_size:
    ↪ int):
    #...
    pass

def forward(self, x: np.array) -> np.array:
    #...
    pass

def backward(self, x: np.array, grad: np.array =
    ↪ np.array([[1]])) -> np.array:
    # don't mind me this week
    pass

```

**Neural Network** We now have all of the required building blocks to create our first neural network! We will implement a neural network also as a module, i.e. as a class.

In the constructor of the class, we can create all layers and activation functions that we need for the network. In the `forward` function the flow of an input  $x$  through the network is then implemented. The return value of this function should be the output of the network.

To make the network more flexible, implement the class such that we can specify the following parameters during initialization:

**input\_size** The number of input neurons

**output\_size** The number of output neurons

**hidden\_sizes** The number of neurons in the hidden layers as a list

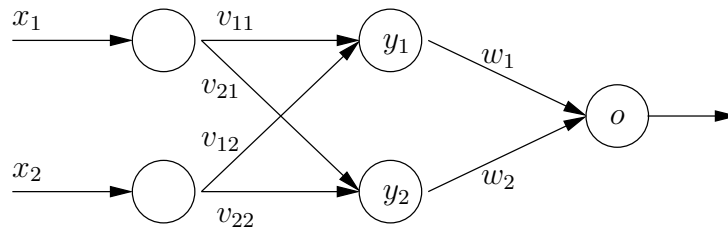
**activation** A class reference of the activation function used for the hidden layers

```
class NeuralNetwork:
    def __init__(self,
                  input_size: int,
                  output_size: int,
                  hidden_sizes: List[int],
                  activation=Sigmoid):
        #...
        pass

    def forward(self, x: np.array) -> None:
        #...
        pass

    def backward(self, x: np.array, grad: np.array =
        → np.array([[1]])) -> Tuple[np.array]:
        # don't mind me this week
        pass
```

**Testing the Implementation** We now have a (hopefully working) Neural Network class. To test this, we will model the network from last week. The network looks like this:



We compute the output and the loss function of the network for input  $x = (1, 1)$  and target label  $t = 0$ . Assume the same initial values as last week:

$$v = \begin{pmatrix} 0.5 & 0.75 \\ 0.25 & 0.25 \end{pmatrix}, w = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

One very convenient thing in Python is that we can just change values of object attributes.

```

if __name__ == "__main__":
    # Network Initialization
    net = NeuralNetwork(2, 1, [2], Sigmoid)

    # Setting the layer weights
    net.layers[0].weights = np.array([[0.5, 0.75], [0.25,
        ↪ 0.25]])
    net.layers[1].weights = np.array([[0.5], [0.5]])

    # Loss
    loss_function = MeanSquaredError()

    # Input
    x = np.array([[1, 1]])
    y = np.array([[0]])

    # Forward Pass
    pred = net.forward(x)

    # Loss Calculation
    loss = loss_function.forward(pred, y)

    print(f"Prediction: {pred}")
    print(f"Loss: {loss}")

```