Prof. Dr. Andreas Hotho, Albin Zehe, Konstantin Kobs
Lehrstuhl für Data Science

11./12.5.2021

2. Assignment in "Machine Learning for Natural Language Processing"

Summer Term 2021

1 General Questions

1. Name two reasons why word embeddings are better suited for most NLP tasks than 1-hot encodings!

   - word embeddings encode word similarity, 1-hot encodings do not
   - the word itself is usually less important than its meaning and its relation to other words

2. Given two word embeddings E and E' for a corpus of words, how could you rate the quality of the embeddings relative to one another? Is there a possibility to give an absolute, global rating for word embeddings?

   Word embeddings can not be rated by themselves, but must be evaluated by their performance when used to solve some task. There are several commonly used tasks used to rate word embeddings:

   - Human Intuition Datasets. These are lists of pairs of words rated by human annotators regarding how similar their meaning is. One example of this is the WS-353 dataset (`http://alfonseca.org/eng/research/wordsim353.html`). Word embeddings are rated by their ability to predict the similarity scores assigned by the annotators. More detail on this soon in the lecture!
   - Evaluation by some classifier that uses word embeddings to represent its input. One task commonly used for this is sentiment analysis. Here, the input sentences are often represented by concatenating the embeddings of their words. If embedding E provides a "better" representation than E', then the same classifier will perform better using E rather than E'.

     Note that this evaluation is very specific to the task! There may be a

task where E clearly outperforms E' and another where E' clearly outperforms E. For example, in sentiment analysis, the words *good* and *bad* are clearly on the opposite ends of the "meaning-scale", while for POS-tagging they are basically equivalent.

2 Neural Networks

2.1 Bias Trick

Formally, a neuron in a neural network is defined by a number of parameters consisting of a weight matrix W and a bias vector b. The output y that is passed to the activation function is then calculated as

$$y = Wx + b.$$

In practice, we want to keep the number of vectorised operations as small as possible, to enable maximum training/prediction speed (remember that GPUs can parallelise the matrix operations). Therefore, these two parameter sets are often combined into one matrix

$$W' = W \| b = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ \vdots & & \ddots & & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} & b_n \end{pmatrix}$$

1. How does the input x need to be modified to enable computing the neurons output y in a single operation?

The input needs to be modified by appending a 1 to it, i.e. $x' = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{pmatrix}$

2. How is the output y calculated using the modified W' and input x'?

$$y = Wx + b = W' \cdot x'$$

3. Proof that this always yields the same result as $Wx + b$!

$$
\begin{aligned}
Wx + b &= \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & & \ddots & \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m + b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m + b_2 \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m + b_n \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
W'x' &= \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ \vdots & & \ddots & & \\ w_{n1} & w_{n2} & \cdots & w_{nm} & b_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m + 1b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m + 1b_2 \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m + 1b_n \end{pmatrix}
\end{aligned}
$$

$$Wx + b = W'x'$$

3 Python

3.1 Installing PyTorch

- Install `python3` on your computer. The most recent version can be found on python.org.[1]
- Install PyTorch on your computer. Usually, this can be done by simply typing `pip3 install torch`. Installation instructions can be found at `https://pytorch.org/get-started/locally/`.
- Optional: If you have a recent Nvidia GPU, you can also install cuda[2] and cuDNN[3] to use your GPU to speed up training. Note that downloading cuDNN requires membership of the Nvidia Developer Program. This membership is free and you only have to fill in a short application.
  After installing cuda and cuDNN, PyTorch should be installable given the suitable installation command they provide on their page.

  Using an AMD GPU with PyTorch is currently not officially supported but in development. *No one here has tested this so far, so proceed at your own risk.*[4]

3.2 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

[1] `https://www.python.org/downloads/`
[2] `https://developer.nvidia.com/cuda-downloads`
[3] `https://developer.nvidia.com/cudnn`
[4] `https://rocm.github.io/pytorch.html`

"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

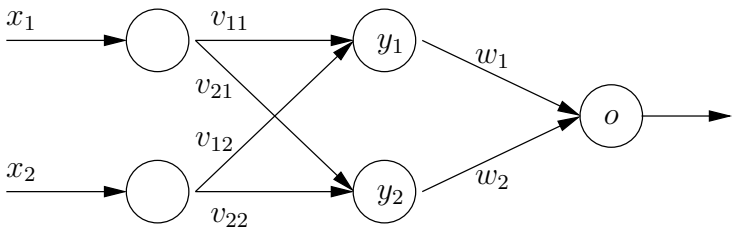
activation A class reference of the activation function used for the hidden layers

```
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}")
```

```python
import numpy as np
from typing import List, Tuple


class Sigmoid:
    def __init__(self):
        pass

    def forward(self, x: np.array) -> np.array:
        return 1 / (1 + np.exp(-x))

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


class MeanSquaredError:
    def __init__(self):
        pass

    def forward(self, y_pred: np.array, y_true:
    ↪  np.array) -> float:
        return np.mean(0.5 * (y_true - y_pred) ** 2)

    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


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

        self.weights = np.random.randn(self.input_size,
        ↪  self.output_size)
```

```
        self.bias = np.zeros((1, self.output_size))

    def forward(self, x: np.array) -> np.array:
        return np.matmul(x, self.weights) + self.bias

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


class NeuralNetwork:
    def __init__(self,
                 input_size: int,
                 output_size: int,
                 hidden_sizes: List[int],
                 activation=Sigmoid):
        s = [input_size] + hidden_sizes + [output_size]

        self.layers = [FullyConnectedLayer(
            s[i], s[i+1]) for i in range(len(s) - 1)]
        self.activation = activation()

    def forward(self, x: np.array) -> None:
        for layer in self.layers[:-1]:
            x = layer.forward(x)
            x = self.activation.forward(x)

        # The last layer should not be using an
         ↪  activation function
        x = self.layers[-1].forward(x)

        return x

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

```python
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}")
```