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

25./26.05.2021

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

Summer Term 2021

1 General Questions

1. Why do we want to vectorise the forward and backward passes of neural networks?

2 Neural Networks

Softmax and Categorical Cross Entropy

In the lecture, you learned that PyTorch's Categorical Cross Entropy Loss function implementation takes the unnormalised outputs of the network (called logits) and implicitly applies Softmax function to obtain a probability distribution that is required in the mathematical definition of Cross Entropy.

Cross Entropy is defined as $L_{CE}(p) = -\sum_{i=1}^{n} y_i \log(p_i)$, where p is the predicted probability distribution, y is the desired output probability distribution, and n is the length of both of these vectors. Assuming that only one class is correct, i.e. y is a one-hot encoded vector, this leads to Categorical Cross Entropy: $L_{CCE}(p) = -\log(p_c)$, where c is the index of the correct output class.

On the other hand, Softmax is defined as $Softmax(o_i) = \frac{e^{o_i}}{\sum_{k=1}^{n} e^{o_k}}$ for an element of the unnormalised output vector o.

To optimise the gradient calculation of the combination of Softmax and Categorical Cross Entropy, PyTorch combines them in one single Module.

In order to understand why implicitly applying the Softmax function is more efficient, calculate the derivatives for the combination of both functions w.r.t. the output of the neural network o (you should do this by hand). Compare this to the derivatives of each step (you are encouraged to do this by hand; you will need the quotient rule).

3 Python

3.1 Implementing Neural Networks Part 2 — The Backward 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.

In the lecture, we have covered vectorised backpropagation in detail, that we also want to use in this exercise for efficiency.

Backward Pass Use this information to implement the `backward` functions for each of the module classes you implemented last week. Each `backward` function gets the input to the function as well as the backpropagating gradient and should output the new gradient for this module. For `FullyConnectedLayer`, return a tuple with the gradient w.r.t. the input, the gradient w.r.t. the weights, and the gradient w.r.t. the bias. `NeuralNetwork` should return a tuple with the gradient w.r.t. the input, a list of gradients w.r.t. the weights of each layer, and a list of gradients w.r.t. the biases of each layer.

Testing the Implementation Apply your backward pass to the network you implemented last week by adding the following code after your forward pass:

```
# Backward Pass
grad = loss_function.backward(pred, y)
grad, W_grads, b_grads = net.backward(x, grad)

print(f"Gradients of the first layer: W1: {W_grads[0]},
 ↪  b1: {b_grads[0]}")
print(f"Gradients of the second layer: W2: {W_grads[1]},
 ↪  b2 {b_grads[1]}")
```

Check that the gradient computed by your network is the same as the one you computed manually.