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

1./2.06.2021

4. Assignment in “Machine Learning for Natural Language Processing”

Summer Term 2021

1 General Questions

1. How are convolutions interpreted in NLP tasks?

> Convolutions in NLP are often interpreted as selective n-grams. Since they are usually computed over the full width of the embeddings and a “height” of about 3 words, the convolutions/filters basically extract three-grams, while having the ability to focus on only the most distinctive n-grams, in contrast to using all of them.

2. What is the effect of pooling layers (max/average pooling) in Convolutional Neural Networks? How often are they commonly used?

> Pooling Layers reduce the size of the input in a pre-defined manner. This means that the number of parameters needed in the following (fully connected) layers is smaller. Pooling layers also do not introduce any additional parameters, as they are purely static operations.
>
> It is common to use a pooling layer after 1 to 3 convolutional layers.

2 Neural Network Hiccups

Dying ReLUs

A frequently used activation function for neural networks is the ReLU-function:

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & \text{else} \end{cases}$$

ReLUs sometimes suffer from the so-called "dying ReLU" problem. In this assignment, you will see what this means and how it can occur.

Assume a neural network with a scalar output, that is, the final layer of the network consists of only one neuron $o = \sum_{i=1}^{n} w_i h_i$, where w are the weights of the layer and h is the output of the previous layer. Let's put a ReLU activation after that layer, to get $y = ReLU(o)$.

We use squared error as the loss function of the network:

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

where t is the true label for the input.

Let $w = \begin{pmatrix} 0.2 & -0.3 & 0.5 \end{pmatrix}$, $h = \begin{pmatrix} 0.1 & 0.5 & 10.0 \end{pmatrix}$ and $t = 0.7$.

Perform the following steps:

1. Compute the gradient $\frac{\partial L}{\partial w}$!
2. Update the weights w using gradient descent with a learning rate of $\lambda = 0.1$
3. Repeat steps (1) and (2) with the updated weights, the input $h = \begin{pmatrix} 0.3 & 0.1 & 1.0 \end{pmatrix}$ and $t = 0.1$.
4. Repeat steps (1) and (2) with the updated weights, the input $h = \begin{pmatrix} 0.5 & 0.25 & 5 \end{pmatrix}$ and $t = 1$.

Describe what you find!

As we already know, the derivative of the square error is

$$\frac{\partial L}{\partial y} = y - t.$$

Then

$$\begin{aligned} \frac{\partial L}{\partial o} &= \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial o} \\ &= (y - t) \cdot \begin{cases} 1, & o > 0 \\ 0, & \text{else} \end{cases} \\ &= \begin{cases} y - t, & o > 0 \\ 0, & \text{else} \end{cases} \end{aligned}$$

For updating the weights:

$$\frac{\partial L}{\partial w} = \begin{pmatrix} \frac{\partial L}{\partial w_1} & \frac{\partial L}{\partial w_2} & \frac{\partial L}{\partial w_3} \end{pmatrix}$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial o} h_i$$

With the given weights and input

$$o = 0.2 \cdot 0.1 + (-0.3) \cdot 0.5 + 0.5 \cdot 10.0 = 0.02 + (-0.15) + 5.0 = 4.87$$

and thus

$$\frac{\partial L}{\partial o} = 4.87 - 0.7 = 4.17.$$

$$\frac{\partial L}{\partial w} = 4.17 \cdot \begin{pmatrix} 0.1 & 0.5 & 10.0 \end{pmatrix} = \begin{pmatrix} 0.417 & 2.085 & 41.7 \end{pmatrix}$$

The updated weights are

$$w' = w - \lambda \cdot \frac{\partial L}{\partial w} = \begin{pmatrix} 0.1583 & -0.5085 & -3.67 \end{pmatrix}$$

Doing the same calculations in the next step, we get

$$\frac{\partial L}{\partial w} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}.$$

The weights stay the same. This also happens in the next step! In fact, we will quite likely never get the neuron to fire again - it has died.

3 Python

3.1 Implementing Neural Networks Part 3 — Dropout

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 Dropout regularisation.

For this, implement a new module (`forward` and `backward` function) that is initialised with a parameter p, denoting the probability that a weight is dropped. Do not forget to scale the resulting weights to make up for the missing weights.

```
class Dropout:
    def __init__(self, p=0.5):
        self.p = p

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

    def backward(self, grad: np.array = np.array([[1]]))
     ↪  -> np.array:
        #...
```

Modify the implementation of the `NeuralNetwork` to include dropout with a specified rate (`dropout:float=0.5` in the constructor) after every hidden layer! Then check that each run of the program will result in different results due to the random cancellation of weights.

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


class Dropout:
    def __init__(self, p=0.5):
        self.p = p

    def forward(self, x: np.array) -> np.array:
        self.mask = np.random.rand(*x.shape) > self.p
        # Scale the mask to even out missing neurons
        x = x * self.mask / self.p
        return x

    def backward(self, grad: np.array = np.array([[1]]))
     ↪  -> np.array:
        # Scale the mask to even out missing neurons
        return grad * self.mask / self.p


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:
        return grad * (self.forward(x) * (1 -
         ↪  self.forward(x)))


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:
        return grad * (y_pred - y_true)


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:
        x_grad = np.matmul(grad, self.weights.T)
        W_grad = np.matmul(x.T, grad)
        b_grad = grad

        return (x_grad, W_grad, b_grad)


class NeuralNetwork:
    def __init__(self,
                 input_size: int,
                 output_size: int,
                 hidden_sizes: List[int],
                 activation=Sigmoid,
                 dropout:float=0.5):
        s = [input_size] + hidden_sizes + [output_size]

        self.layers = [FullyConnectedLayer(s[i], s[i+1])
         ↪   for i in range(len(s) - 1)]
        self.dropouts = [Dropout(dropout) for i in
         ↪   range(len(s) - 2)]
        self.activation = activation()

    def forward(self, x: np.array) -> None:
        self.layer_inputs = []
        self.activ_inputs = []

        for layer, dropout in zip(self.layers[:-1],
         ↪   self.dropouts):
            self.layer_inputs.append(x)
            x = layer.forward(x)
            self.activ_inputs.append(x)
            x = self.activation.forward(x)

            # Dropout Layer
            x = dropout.forward(x)

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

```
            return x

        def backward(self, x: np.array, grad: np.array =
         ↪  np.array([[1]])) -> Tuple[np.array]:
            W_grads = []
            b_grads = []

            grad, W_grad, b_grad =
             ↪  self.layers[-1].backward(self.layer_inputs[-1],
             ↪  grad)
            W_grads.append(W_grad)
            b_grads.append(b_grad)

            for i in
             ↪  reversed(range(len(self.activ_inputs))):
                # Dropout Layer
                grad = self.dropouts[i].backward(grad)

                grad =
                 ↪  self.activation.backward(self.activ_inputs[i],
                 ↪  grad)
                grad, W_grad, b_grad =
                 ↪  self.layers[i].backward(self.layer_inputs[i],
                 ↪  grad)
                W_grads.append(W_grad)
                b_grads.append(b_grad)

            return grad, list(reversed(W_grads)),
             ↪  list(reversed(b_grads))


if __name__ == "__main__":
    # Network Initialization (with Dropout)
    net = NeuralNetwork(2, 1, [2], Sigmoid, dropout=0.5)

    # 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]])
```

```python
# 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}")

# 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]}")
```