Prof. Dr. Andreas Hotho, Albin Zehe, Jonas Kaiser
Data Science Chair

05./06.06.2024

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

Summer Term 2024

1 General Questions

1. How can we apply a CNN model to text?
   a) What is the input to the network?
   b) How do we work with different text lengths in one batch and across batches?
   c) What are the sizes of the CNN kernels?

? Something to think about
Could we use **mean**-over-time pooling in the TextCNN by Kim? What problem can arise in situations where we have texts of different length?

We can use the ideas of the TextCNN by Kim.

a) The input usually are word embeddings of the input words, concatenated in order to form a two-dimensional matrix with shape (word embedding dimensions, length of the input sentence with optional padding).

b) Texts of different lengths in one batch are padded to the same length. Due to the max-over-time pooling, we can work with sequences of different lengths across batches.

c) The kernels have a size of (word embedding dimensions, x), where x is a number such as 3, 5, or 7. The larger x, the more context the CNN grasps. However, larger kernels mean more parameters. To get some intuition: An x of 2 should be sufficient in most cases to find negations, e.g. "I am **not amused**", as we only need a context of 2 words to find these kinds of negation. Mostly, x is odd as we then have the same context size before and after a word when performing the convolution.

> Using mean-over-time pooling would mean that we average the activations across the whole text sequence. Having a batch with short and long texts, we would still need to apply padding to make the texts the same length. Using zero-padding, the activations after the convolutions would be zero for most of the padded parts. The average activation would then get very small, which could reduce the performance of the feed-forward layer.

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 Convolutions

3.1 Manual Convolution

Given the matrix

$$M = \begin{pmatrix} 0 & 6 & 1 & 1 & 6 \\ 7 & 9 & 3 & 7 & 7 \\ 3 & 5 & 3 & 8 & 3 \\ 8 & 6 & 8 & 0 & 2 \\ 4 & 3 & 2 & 9 & 1 \end{pmatrix}$$

and a kernel

$$k = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Calculate the convolution $C = M * k$ using zero padding to make the output matrix C have the same size as M and a stride of 1! Is there a difference between a convolution and a cross correlation in this case?

$$
\begin{aligned}
C_{1,1} &= M_{1,1} \cdot w_{2,2} + M_{1,2} \cdot w_{2,3} + M_{2,1} \cdot w_{3,2} + M_{2,2} \cdot w_{3,3} \\
&= 0 + 6 + 7 + 9 = 22 \\
C_{1,2} &= M_{1,1} \cdot w_{2,1} + M_{1,2} \cdot w_{2,2} + M_{1,3} \cdot w_{3,3} + M_{2,1} \cdot w_{3,1} + M_{2,2} \cdot w_{3,2} + M_{2,3} \cdot w_{3,3} \\
&= 0 + 6 \cdot (-8) + 1 + 7 + 9 + 3 = -28 \\
&\vdots
\end{aligned}
$$

$$
C = \begin{pmatrix} 22 & -28 & 18 & 16 & -33 \\ -33 & -44 & 16 & -24 & -31 \\ 11 & 7 & 22 & -31 & 0 \\ -43 & -12 & -28 & 36 & 5 \\ -15 & 4 & 10 & -59 & 3 \end{pmatrix}
$$

3.2 Rotated Convolution

When deriving the formula for backpropagation over a convolutional layer, a substitution of a form similar to this is done:

$$
\begin{aligned}
\frac{\partial L}{\partial x_{a,b}} &= \sum_{m=-l'}^{l'} \sum_{n=-l'}^{l'} \delta_{h_{a+m,b+n}} w_{-m+l'+1,-n+l'+1} \\
&= \delta_{h_{a-l':a+l',b-l':b+l'}} * rot_{180}(w)
\end{aligned}
$$

where x is the input matrix, $w \in \mathbb{R}^{r \times r}$ is the kernel matrix and $l' := \left\lfloor \frac{r}{2} \right\rfloor$ is half the width/height of the kernel matrix.

Show that the double sum on the left-hand side can indeed be expressed by the convolution with rotated kernel on the right-hand side! It is not necessary to mathematically prove this relation. There are other ways, such as tracking the transformation of single indices.

Here is a “math-version” of this transformation:

$$
\begin{aligned}
&\left(\delta_{h_{a-l'+i,b-l'+j}}\right)_{0\leq i,j\leq r-1} * rot_{180}\left(\left(w_{i,j}\right)_{0\leq i,j\leq r-1}\right) \\
&= \left(\delta_{h_{a-l'+i,b-l'+j}}\right)_{0\leq i,j\leq r-1} * \left(w_{r-i,r-j}\right)_{0\leq i,j\leq r-1} \\
&\underset{\text{def}}{=} \sum_{i=0}^{r-1}\sum_{j=0}^{r-1} \delta_{h_{a-l'+i,b-l'+j}} w_{r-1-i,r-1-j} \\
&= \sum_{i=-l'}^{l'}\sum_{j=-l'}^{l'} \delta_{h_{a+i,b+j}} w_{l'-i+1,l'-j+1}
\end{aligned}
$$

4 Python

4.1 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 (`p: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))):
```

```python
            # 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]])

    # 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
# 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]}")
```