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

15./16.06.2021

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

Summer Term 2021

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 Convolutions

2.1 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?

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

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

2.2 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:

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

3 Python

3.1 Implementing Neural Networks Part 4 — Optimisers

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 adapt the optimisers from the first assignment to train your neural network.

1. In the first assignment, you have already implemented some common optimisers for neural networks (SGD, Nesterov Momentum and Adam).

   In this exercise, you will add optimizer functions for SGD, simple Momentum, and Adam to our framework. Add classes `SGD`, `Momentum`, and `Adam` to your code. Their constructor should get the instantiated `NeuralNetwork` object and the necessary hyper-parameters such as the learning rate. The classes should provide a method called `update` that gets the weight and bias gradients from the gradient calculation. This method takes one step of the corresponding optimizer and updated the weights and biases of the given neural network model.

```
class SGD:
    def __init__(self, nn:object, lr:float):
        #...
        pass
    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        #...
        pass


class Momentum:
    def __init__(self, nn: object, lr: float, mu:
     ↪  float):
        #...
        pass

    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        #...
        pass


class Adam:
    def __init__(self, nn: object, lr: float, beta1:
     ↪  float, beta2: float):
        #...
        pass

    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        #...
        pass
```

2. Train the neural network you have implemented so far using these optimisers! For example, you can use the network to learn the `XOR`-Function:

x	y	`XOR(x,y)`
0	0	0
0	1	1
1	0	1
1	1	0

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


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))



class SGD:
    def __init__(self, nn:object, lr:float):
        self.nn = nn
        self.lr = lr

    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        for i in range(len(self.nn.layers)):
            self.nn.layers[i].weights -= self.lr *
             ↪  W_grads[i]
            self.nn.layers[i].bias -= self.lr *
             ↪  b_grads[i]


class Momentum:
    def __init__(self, nn: object, lr: float, mu:
     ↪  float):
        self.v_W = [0] * len(nn.layers)
        self.v_b = [0] * len(nn.layers)

        self.nn = nn
        self.lr = lr
        self.mu = mu

    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        for i in range(len(self.nn.layers)):
            self.v_W[i] = self.mu * self.v_W[i] -
             ↪  self.lr * W_grads[i]
            self.v_b[i] = self.mu * self.v_b[i] -
             ↪  self.lr * b_grads[i]

            self.nn.layers[i].weights += self.v_W[i]
            self.nn.layers[i].bias += self.v_b[i]


class Adam:
    def __init__(self, nn: object, lr: float, beta1:
     ↪  float, beta2: float):
        self.m_W = [0] * len(nn.layers)
        self.m_b = [0] * len(nn.layers)

        self.v_W = [0] * len(nn.layers)
        self.v_b = [0] * len(nn.layers)

        self.nn = nn
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2

    def update(self, W_grads: List[np.array], b_grads:
     ↪  List[np.array]) -> None:
        for i in range(len(self.nn.layers)):
            self.m_W[i] = self.beta1 * self.m_W[i] + (1
             ↪  - self.beta1) * W_grads[i]
            self.m_b[i] = self.beta1 * self.m_b[i] + (1
             ↪  - self.beta1) * b_grads[i]

            self.v_W[i] = self.beta2 * self.v_W[i] + (1
             ↪  - self.beta2) * W_grads[i]**2
            self.v_b[i] = self.beta2 * self.v_b[i] + (1
             ↪  - self.beta2) * b_grads[i]**2

            self.nn.layers[i].weights -= self.lr *
             ↪  self.m_W[i] / (np.sqrt(self.v_W[i]) +
             ↪  1e-8)
```

```
                self.nn.layers[i].bias -= self.lr *
                 ↪  self.v_b[i] / (np.sqrt(self.v_b[i]) +
                 ↪  1e-8)


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

    # Setting the layer weights for reproduction
     ↪  purposes
    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()

    # Optimizer
    optimizer = SGD(net, lr=0.1)
    # optimizer = Momentum(net, lr=0.1, mu=0.0)
    # optimizer = Adam(net, 0.001, 0.9, 0.999)

    # XOR Dataset
    inputs = [
        (np.array([[0, 0]]), np.array([0])),
        (np.array([[0, 1]]), np.array([1])),
        (np.array([[1, 0]]), np.array([1])),
        (np.array([[0, 0]]), np.array([0]))
    ]

    epochs = 50000

    # tqdm (you need to install it using `pip3 install
     ↪  tqdm`)
    # will show a progress bar for the loop
    for epoch in tqdm(range(epochs)):
        # It is always a good idea to shuffle the
         ↪  dataset
        np.random.shuffle(inputs)

        for x, y in inputs:
            # Forward Pass
            pred = net.forward(x)

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

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

            optimizer.update(W_grads, b_grads)


    # Test that the network has learned something
    for x, y in inputs:
        # Forward Pass
        pred = net.forward(x)

        print(f"Input: {x}, Output: {pred}, Desired
         ↪  Output: {y}")
```