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

12./13.06.2024

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

Summer Term 2024

1 General Questions

1. 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. 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.

? Something to think about

3. Can we implement a fully connected layer in a neural network by using a convolutional layer? If so, how?

   Is the opposite possible, i.e. can we imitate the behaviour of a convolutional layer using a fully connected layer?

Yes, a convolutional layer can implement a fully connected (dense) layer. The important steps are:

- Use a kernel size that matches the input's dimensions.
- Set the stride to 1 and do not use padding. This way, we only perform one convolution for each filter over the entire input.
- Use a number of filters equal to the number of neurons in the fully connected layer.

This way, each filter effectively connects to every input element and produces one output, replicating the behaviour of a fully connected layer.

While doing the opposite is technically possible, we cannot do it without losing some of the advantages of the convolutional layer. For example, a dense layer can only process a fixed number of inputs. While we can pad smaller input sizes, we cannot apply our layers to larger ones. Even if we were to accept this limitation, we'd still face problems due to the massive amount of parameters required to model a convolution operation via a fully connected layer.

2 Backpropagation in Convolutional Neural Networks

Given the matrix

$$M = \begin{pmatrix} 1 & 1 & 1 & 2 & 3 \\ 1 & 1 & 1 & 2 & 3 \\ 1 & 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix}$$

and a kernel

$$k = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix},$$

assuming $L = \sum_{row=1}^{4} \sum_{col=1}^{3} O_{row,col}$ and $O = M * k$, calculate $\frac{\delta L}{\delta O}, \frac{\delta L}{\delta k}, \frac{\delta L}{\delta M}$. You do not need to apply the individual equations. For clarity, in this case " $*$ " refers to the actual convolution and not just cross-correlation.

> Note, that in the backward pass we use the cross-correlation, since we applied a convolution in the foward pass (represented via $\star$ in the equations below).
>
> $$\frac{\delta L}{\delta O} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$
>
> $$\frac{\delta L}{\delta k} = M \star \frac{\delta L}{\delta O} = \begin{pmatrix} 1 & 1 & 1 & 2 & 3 \\ 1 & 1 & 1 & 2 & 3 \\ 1 & 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix} \star \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 15 & 18 & 25 \\ 21 & 23 & 28 \\ 30 & 31 & 34 \end{pmatrix}$$
>
> $$L' = pad(\frac{\delta L}{\delta O}, 2, 0) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\frac{\delta L}{\delta M} = L' \star k = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \star \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & -1 & -1 \\ 3 & 3 & 0 & -3 & -3 \\ 4 & 4 & 0 & -4 & -4 \\ 4 & 4 & 0 & -4 & -4 \\ 3 & 3 & 0 & -3 & -3 \\ 1 & 1 & 0 & -1 & -1 \end{pmatrix}$$

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.m_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}")
```