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

22./23.05.2024

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

Summer Term 2024

1 General Questions

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

   There are many optimised strategies for matrix and vector operations such as multiplication (see, e.g. `https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm#Sub-cubic_algorithms` for algorithms that make multiplication more efficient). Additionally, GPUs are optimised for these tasks and highly parallelisable. This way, we can speed up the training and application of the neural network. This becomes handy when working with large datasets or very deep neural networks.

2. Explain why using dropout during training of a neural network can be considered as training and ensemble of models. Why does this help combat overfitting? Why might a model trained with dropout perform better on validation data than during training?

   How does dropout work like an ensemble?

   When you apply dropout with a probability p, each neuron has a p chance of being "dropped out" or set to zero during forward propagation. This means that, during each update, you’re essentially training a different sub-network with a random subset of the original n neurons which could be viewed as samller, but fully functional neural network

   Why does this help combat overfitting?

   By training an ensemble of sub-networks, you’re essentially averaging out the noise and variability in the data. Each sub-network is forced to learn a robust representation of the input data, as it cannot rely on specific neurons at all times. This avoids an overreliance on indiviudal "patterns" that may be present in the training data but aren't useful for other datasets.

Why might a model trained with dropout perform better on validation data?

During training, dropout is active, and neurons are randomly dropped, which adds noise and variability to the training process. However, during validation (and inference), dropout is turned off, and all neurons are used, typically scaled by the dropout rate to account for the missing neurons during training. This change can lead to better performance on validation data because the model is effectively more powerful and can leverage the more robust features learned during training. By using all neurons we are essentially usingg the entire "ensemble" at once, rather than just a single sub-network

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 $p_i = Softmax(o_i) = \frac{e^{o_i}}{\sum_{k=1}^{n} e^{o_k}}$ 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 individually (you are encouraged to do this by hand; you will need the quotient rule).

First, we calculate the derivatives for each function separately. You can find a step-by-step calculation for Softmax here: `https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/`. Given a Softmax output index i and an input index j, we can calculate the gradient flowing through the Softmax by

$$\frac{\partial Softmax(o_i)}{\partial o_j} = \begin{cases} Softmax(o_i) \cdot (1 - Softmax(o_j)) & \text{, if } i = j \\ -Softmax(o_j) \cdot Softmax(o_i) & \text{, if } i \neq j \end{cases} .$$

For Categorical Cross Entropy, we have to consider the case that we calculate the derivative w.r.t. the correct class index c or to another index. We then get

$$\frac{\partial L_{CCE}}{\partial p_j} = \begin{cases} -\frac{1}{p_j} & j = c \\ 0 & j \neq c \end{cases} .$$

Now, we combine both functions first and calculate the derivative directly. We can calculate the loss directly from the unnormalised output vector o:

$$L_{CCE} = -\log\left(\frac{e^{o_c}}{\sum_{i=1}^{n} e^{o_i}}\right) .$$

We can now rewrite this as

$$L_{CCE} = -\log(e^{o_c}) + \log\left(\sum_{i=1}^{n} e^{o_i}\right) = -o_c + \log\left(\sum_{i=1}^{n} e^{o_i}\right) .$$

Now, the calculation of the derivative depends on whether we derive w.r.t. the correct class output o_c or to another output.

$$\frac{\partial L_{CCE}}{\partial o_j} = \begin{cases} -1 + \frac{1}{\sum_{i=1}^{n} e^{o_i}} \cdot e^{o_j} = -1 + Softmax(o_j) & \text{, if } j = c \\ \frac{1}{\sum_{i=1}^{n} e^{o_i}} \cdot e^{o_j} = Softmax(o_j) & \text{, if } j \neq c \end{cases}$$

Combining both functions requires the use of a Softmax function, which in turn simplifies gradient calculation. We can reuse the results of the Softmax for the gradients. Therefore, PyTorch combines both steps: to make sure, that this easy calculation is possible, the Softmax is applied inside of the CrossEntropy Loss Module.

3 Python

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.

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


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]])) -> Tuple[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):
```

```
        s = [input_size] + hidden_sizes + [output_size]

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

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

        for layer in self.layers[:-1]:
            self.layer_inputs.append(x)
            x = layer.forward(x)
            self.activ_inputs.append(x)
            x = self.activation.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))):
            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
    net = NeuralNetwork(2, 1, [2], Sigmoid)

    # 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}")

    # 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]}")
```

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.