Advanced Introduction to C++, Scientific Computing and Machine Learning

SoSe 24 | 08.07.2024
C. Gros, D. Nevermann

Exercise Sheet #13

Note This sheet is optional and will not count towards your final grade.

Rock-Paper-Scissors World Cup The tournament, where your projects clients will compete in the ultimate rock-paper-scissors world cup, will take place July, 16th 2024 in lecture.

Exam The exam will take place July 24th, 2024 at 10:15h in Phys__.102.

Problem 1 (*PyTorch: Perceptron Training*)

A simple neural network is a perceptron (link) with just two layers: the input and output layer. We here want to train a perceptron to resemble a logical AND gate with the truth table

x_1	x_2	$\mathrm{AND}(x_1, x_2)$
0	0	0
1	0	0
0	1	0
1	1	1

The perceptron receives a two-dimensional input vector

$$\vec{x} \in \{(0,0), (0,1), (1,0), (1,1)\}$$

The perceptron output is obtained through

$$y = \sigma(x_1 w_1 + x_2 w_2 + b),$$

where $\vec{w} = (w_1, w_2)$ is the weight vector, b is a bias and

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

is the sigmoid transfer function. Implement and train the perceptron in PyTorch:

- Define a tensor for the inputs. Further, define a tensor for the weights and a tensor for the bias. Initialize the weights and the bias randomly. Make sure to define them using `requires_grad=True`.

- Write a function that computes the forward pass.

Next we want to train the perceptron:

- Define a tensor with the correct outputs for the given inputs (our training data).
- Write a function that computes a loss

  $$L = (y - \hat{y})^2,$$

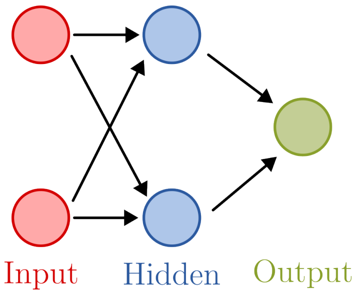
  where y is the correct output and $\hat{y}$ is the obtained value through the forward pass.
- Back propagate the loss and update the weights and the bias using gradient decent with a learning rate $\eta = 0.1$

  $$w_i \to w_i - \eta(\nabla w)_i, \quad b \to b - \eta \nabla b.$$

  Do this for 10000 epochs and try out the trained perceptron.

Problem 2 (*PyTorch: The XOR Problem and Optimizers*)

In lecture, you discussed the XOR-problem (link), i.e. that XOR can not be solved by a linear classifier, as the XOR gate is not linearly separable. This problem can be solved, however, by introducing a hidden layer.
The code given below (and here for download) implements a simple neural network with one hidden layer that replicates an XOR gate. Here is a sketch of the architecture:



In the `MyLayer` class, the `update` method explicitly implements parameter tweaking via gradient descent. PyTorch, however, provides optimizers that efficiently implement parameter optimization.
Your task is to modify the code such that the explicit training via gradient descent is replaced by an optimizer from `torch.optim`. A good choice is `torch.optim.Adam`.

```
#!/usr/bin/env python3

import torch

torch.manual_seed(42)

#
# tanh layer
#
class MyLayer(torch.nn.Module):    # inheritance
    def __init__(self, dim1, dim2):  # constructor
        super().__init__()
        self.weights = torch.randn(dim1,dim2,requires_grad=True)
        self.bias    = torch.randn(dim1,requires_grad=True)

    def forward(self, x):            # define forward pass
        return torch.tanh(torch.matmul(self.weights,x)-self.bias)

    def update(self, eps):   # updating weights / bias
        with torch.no_grad():
            self.weights -= eps*self.weights.grad
            self.bias    -= eps*self.bias.grad
            self.weights.grad = None
            self.bias.grad    = None

#
# main
#
dimOutput      = 1            # only 1 implemented
dimHidden      = 2
dimInput       = 2            # only 2 implemented
nEpoch         = 4000
learningRate   = 4.0e-2
myLayerObject  = MyLayer(dimHidden,dimInput)   # instanstiation
myOutputObject = MyLayer(1,dimHidden)


# XOR for 2 inputs
booleanInput = torch.tensor([ [ 1.0, 1.0],
                              [ 1.0,-1.0],
                              [-1.0, 1.0],
                              [-1.0,-1.0] ])

booleanValue = torch.tensor([ [-1.0],
                              [ 1.0],
                              [ 1.0],
                              [-1.0] ])

print(booleanInput)
print(booleanValue)

#
# training loop
#
for iIter in range(nEpoch):                   # trainning loop
#
    thisInput  = booleanInput[iIter%4]
    thisTarget = booleanValue[iIter%4]
#
    hidden = myLayerObject(thisInput)          # forward pass (implicit)
    output = myOutputObject(hidden)
```

```
    loss   = (output-thisTarget).pow(2).sum()  # generic loss function

    if iIter % 100 == 0:
    print(loss.item())

    loss.backward()                              # backward pass

    myLayerObject.update(learningRate)    # gradients have
    myOutputObject.update(learningRate)   # been summed up

# end of training

for input in booleanInput:
    hidden = myLayerObject(input)
    print(f"{input}: {myOutputObject(hidden)}")
```