Exercise Sheet #12

Deadline: 08.07.2024, 12:00h

Exam If you have not signed up for the exam yet, you have the chance to do so until the 17th of July 2024 via an informal email to
nevermann@itp.uni-frankfurt.de.
You do not have to sign up via LSF.
The exam will consist of small problems where you mostly have to correct, extent or determine the output of a `C++` code or a numerical algorithm. Example problem:

Korrigieren Sie die markierte Zeile, sodass das Programm funktioniert.

```cpp
#include <iostream>

int main() {
    cout << "what is wrong?" << endl;      // (*)
    return 0;
}
```

Problems where you have to write a small program yourself may also occur.

PyTorch Python is the dominant language used in machine learning applications and therein PyTorch is the most popular machine learning framework. This sheet will serve as a small introduction to PyTorch.
Start by installing PyTorch via `pip install torch`.[1] Then test your installation by running a small script

```python
import torch
x = torch.rand(5, 3)
print(x)
```

While the following problems will give you some guidance, it will be necessary to consult the documentation.

[1] Or use `pip3`, depending on your installation.

Problem 1 (*PyTorch: Tensors*) (10 points)

The main building block in PyTorch are tensors. From a neural networks input to the weights and hidden layers to the output: everything is stored in a tensor object. You can think of a tensor as a multidimensional array. To create a basic tensor from a Python list, use

```
import torch
x = torch.tensor([1, 2, 3])
```

PyTorch is a Python library, but uses under the hood compiled `C++` to speed up the computations. To take full advantage of that, tensor manipulation should be performed using PyTorch's build functions.
Play around with tensors and operations on them.

- Create different tensors (preferably using PyTorch tensor creation routines). Create

  (1) a $3 \times 3 \times 3$ tensor filled with ones,

  (2) a one dimensional tensor containing the numbers from 0 to 26,

  (3) a $3 \times 3 \times 3$ tensor filled with random values between 0 and 1.

- We want to add the tensors (1) and (2), but they are of different shape. Reshape tensor (2) to obtain a $3 \times 3 \times 3$ tensor and add the two tensors. Then multiply element wise the resulting tensor by the tensor (3).

- Create a tensor containing all the elements of the tensor from the previous point but in a one dimensional tensor.

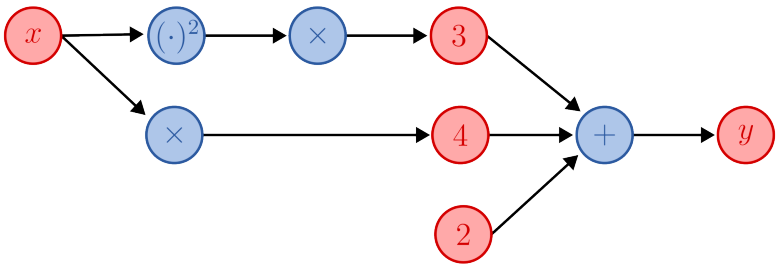
- Like Numpy arrays, PyTorch tensors support smart indexing. Set all negative elements in the tensor from the previous point to zero.

Problem 2 (*PyTorch: Autograd and Computational Graph*) (10 points)

PyTorch can compute derivatives using its `autograd` module. Using a simple example we will see how this works.
Consider the function

$$y = f(x) = 3x^2 + 4x + 2, \qquad x \in \mathbb{R}.$$

PyTorch will, once we command it to do so via the keyword argument `require_grad=True` in the tensor creation function, start to record all operations performed on a tensor and from that construct a *computational graph*. A computational graph is a directed graph containing two different types of nodes: operation nodes (blue) and variable nodes (red). The computational graph of our example function looks like this:



Traversing the graph in the direction of the arrows gives us the *forward pass*. For each operation node PyTorch knows the respective derivative. Traversing the graph backwards while multiplying the derivatives gives us, using the chain rule, the derivative $\partial_x y$. This is in essence *back propagation*. For a deep dive check out this blog article.

- First compute the forward pass for the input value $x = 3$ by hand. Then, also by hand, compute the derivative $\partial_x y(3)$.
- Next, verify your results using PyTorch. Follow these steps:
  (1) Create a scalar tensor `x` with the value 3. Make sure to pass `require_grad=True` to build a computational graph.
  (2) Perform the necessary operations to `x` to end up at y storing the output in a variable `y`, that is perform the forward pass, and compare with your analytical result.
  (3) Call `y.backward()` to do the back propagation and read the back propagated gradient at x using `x.grad`. Compare with your analytical result.