Lab 2 - Basics of Autograd in Python (and first overview of *backpropagation*)

What did we see last time?

- PyTorch basics
- Construction of a multilayer perceptron using PyTorch API

What caught our attention?

Differentiation in PyTorch

PyTorch is built with support for differentiation in mind. In the end, Deep Learning (for now) is all about differentiation and building cascades of differentiable function into complicated multilayer deep neural networks.

Essentially, all PyTorch built-ins support differentiability (unless the function is not differentiable, of course). Today we will see how to compute derivatives in PyTorch. Also, we will learn how to create differentiable modules using PyTorch APIs.

Notation and recall

1. **Function** $f : \mathbb{R} \to \mathbb{R}$, given $x \in \mathbb{R}$, derivative is $\frac{\partial f}{\partial x}$
2. **Scalar function** $f : \mathbb{R}^d \to \mathbb{R}$, we have a vector $\mathbf{x} \in \mathbb{R}^d = (x_1, \dots, x_d)$, we calculate the derivative of f w.r.t. each of the dimensions of $\mathbf{x}$ and obtain the gradient $\nabla_f = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_d})$
3. **Vector function** $f : \mathbb{R}^d \to \mathbb{R}^k$, given $\mathbf{x}$, we have $f(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_k(\mathbf{x}))$, hence we can calculate k gradients which we can gather in the Jacobian: $J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial x_1} & \cdots & \frac{\partial f_k}{\partial x_d} \end{pmatrix} \in \mathbb{R}^{d \times k}$

grad functionality

"Under-the-hood", each PT Tensor has an attribute `requires_grad`

In [1]:
```python
import torch

x = torch.rand(3,3)

x
```

Out[1]:
```
tensor([[0.3195, 0.1809, 0.9280],
        [0.9200, 0.1693, 0.1443],
        [0.1661, 0.4556, 0.7121]])
```

In [2]:
```python
x.requires_grad
```

Out[2]:
```
False
```

We can manually set this to `True` or create directly a Tensor supporting grad.

In [3]:
```python
x.requires_grad = True

x
```

Out[3]:
```
tensor([[0.3195, 0.1809, 0.9280],
        [0.9200, 0.1693, 0.1443],
        [0.1661, 0.4556, 0.7121]], requires_grad=True)
```

In [4]:
```python
torch.rand(3, 3, requires_grad=True)
```

Out[4]:
```
tensor([[0.9883, 0.7529, 0.9830],
        [0.8041, 0.5193, 0.2135],
        [0.3283, 0.3054, 0.3310]], requires_grad=True)
```

Case 1

Suppose we are in case 1.: $f : \mathbb{R} \to \mathbb{R}$.

For instance, $f(x) = x^2$.

We could apply f to a singleton tensor and calculate the derivative.

We expect the derivative to be... ?

In [5]:
```python
x = torch.rand(1, requires_grad=True)

print("x:", x)

y = x**2

print("y:", y)
```

```
x: tensor([0.8687], requires_grad=True)
y: tensor([0.7546], grad_fn=<PowBackward0>)
```

To calculate the gradient, we call `backward()` on the Tensor. Which one, `x` or `y`?

In [7]:
```python
y.backward()
```

We can inspect the gradient of x by accessing its grad attribute:

In [8]:
```python
x.grad
```

Out[8]:
```
tensor([1.7374])
```

Let's check that it's correct...

In [9]:
```python
x.grad == 2*x
```

Out[9]:
```
tensor([True])
```

Notice that, when there's no gradient, it is automatically set to `None` to save memory

In [10]:
```python
torch.rand(3,3).grad is None
```

Out[10]:
```
True
```

Case 2 (scalar function)

We can use the same `.backward()` call to get the gradient of a scalar function.

Now x will be a vector (or a matrix, it doesn't really matter for our case) and we will apply to it a function which returns a single scalar.

One example may be $f(\mathbf{x}) = \sum_{i=1}^{d} x_i$.

Q: What is the gradient we expect to obtain? A vector of ones

In [3]:
```python
x = torch.rand([5], requires_grad=True)

y = x.sum()

y.backward()

x.grad
```

Out[3]:
```
tensor([1., 1., 1., 1., 1.])
```

Case 3 (vector function)

Unfortunately, the backward computation of the gradient is not directly capable of calculating the gradient for a vector of values, but only for a single scalar. So just 1 node as output!

If we wanted to compute the gradient on a vector function, what could we do?

1. There exist a forward differentiation, which is not though present in PT, which lets us calculate derivative of vector functions with one-dimensional input.
2. Using PT backward functionality in a loop through all the outputs (you do it one component at the time in a for loop storing all the gradients)
```
# d is dimension of inputs
# k is dimension of outputs
outputs = model(inputs)
gradients = torch.empty((d, k)) # create empty d x k matrix
for i in range(k):
    outputs[i].backward()
    gradients[:, i] = inputs.grad
    inputs.grad = None
```

Q: Why is really the backward differentiation (and not the forward) useful for our case?

It is useful because we have a model whose ouput gets passed through a loss function $\mathcal{L}$, which is always a scalar. So we can see the process as a composite function $l = \mathcal{L}(f(X))$, where f is our generic Machine Learning model. For deep learning, what we need to calculate is $\partial \mathcal{L} / \partial \Theta$ where Θ are the parameters of our model (in an MLP, that is the collection of weights and biases).

So we don't actually need to compute jacobians.

Composition of functions

We can use also `backward` to compute the gradient of a composition of functions. For our objective, it will be very useful to think in terms of computational graph.

We can view $y = g(f(x))$ as

We might extend this and add a hidden node z between f and g

Supposing $f(x) = log(x)$ and $g(x) = x^2$, we can reproduce this example in PyTorch.

Q

- What we expect to get from $\partial g/\partial z$? $2z$ which is equal to $2\log(x)$
- And from $\partial f/\partial x$? $1/x$
- And from $\partial g/\partial x$? $2\log(x)/x$
- More specifically, what technique do we use to calculate this final gradient? The chain rule ($\partial g/\partial x = \partial g/\partial z \cdot \partial f/\partial x$)

In [13]:
```
x = torch.rand(1, requires_grad=True)

print("x:", x, "\n")

z = x.log()

y = z**2

print("y:", y, "\n")
```

```
x: tensor([0.9460], requires_grad=True)

y: tensor([0.0031], grad_fn=<PowBackward0>)
```

by printing `y`, we can see that the tensor has a specific gradient function attached.

Let us now compute the gradient...

In [14]:
```
y.backward()

print("gradient of x:", x.grad, "\nQ:(gradient of x w.r.t. what?)")# it is the gradient of g wrt x=0.69
```

```
gradient of x: tensor([-0.1174])
Q:(gradient of x w.r.t. what?)
```

Let us access $\partial g/\partial z$

In [15]:
```
## your code here
z.grad #PT doesn't store the data for the gradient: CAREFULL for gradient of intermidiate stuff!!
# you can use backwords-hooks!
z.grad
```

```
/home/zullich/miniconda3/envs/lot/lib/python3.8/site-packages/torch/_tensor.py:1013: UserWarning: The .grad attribu
te of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.
backward(). If you indeed want the .grad field to be populated for a non-leaf Tensor, use .retain_grad() on the non
-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See githu
b.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at  /opt/conda/conda-bld/pytorch_1634
272128894/work/build/aten/src/ATen/core/TensorBody.h:417.)
  return self._grad
```

Note: to store gradients of intermediate computations, we can call `.retain_grad()` on the intermediate node. Example:

```
z.retain_grad()

y.backward()

z.grad -> now it won't be None
```

A more complicated example

In [16]:
```
# Tensor != tensor
# Tensor is the constructor of a class
# With tensor we can specify the shape and it populates the tensor randomly
x_1 = torch.tensor([3.0], requires_grad=True)

x_2 = torch.tensor([2.0], requires_grad=True)

print("x_1:  ", x_1)
print("x_2:  ", x_2)
```

```
x_1:   tensor([3.], requires_grad=True)
x_2:   tensor([2.], requires_grad=True)
```

Construct `c`, calculate the gradient and access it for both `x_1` and `x_2`

In [18]:
```
c = x_1.cos() * x_2.log()
c.backward()
print(x_1.grad)
print(x_2.grad)
```

```
tensor([-0.0978])
tensor([-0.4950])
```

Gradient accumulation

Let us see another feature of torch differentiation functionalities.

We can call `backward()` multiple times; let us see what happens.

In [19]:
```
## repeat the computation for c... : they are doubled values! Why? Gradient gets accumulated!!
c = x_1.cos() * x_2.log()
c.backward()
print(x_1.grad, x_2.grad)
```

```
tensor([-0.1956]) tensor([-0.9900])
```

Q: what is happening? Why the gradient is not the same?

They doubled. PyTorch continues to accumulate (i.e., sum) the gradients. If we want to reset the gradient, we must set it to None

```
x_1.grad = None
x_2.grad = None
```

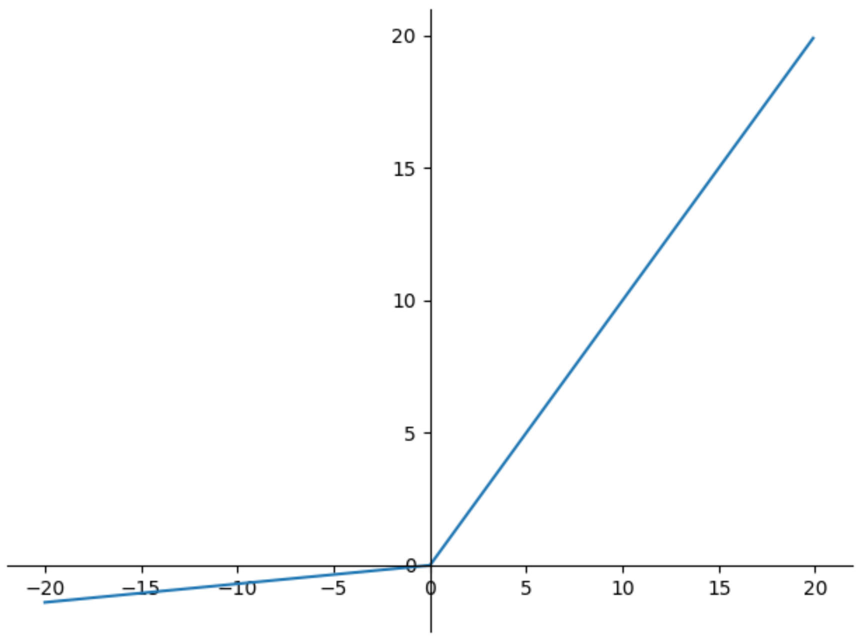
In [40]:
```
# we need to set the var to None
x_1.grad = None
x_2.grad = None
```

Building a custom, non-parametric PyTorch module

Basically, we want to create a module which is not controlled by any parameter, be it trainable or non-trainable.

As an example, we might have the **Leaky ReLU**, an activation function which can be used in place of the more-known ReLU.

$$\text{LeakyReLU} = \max\{0.01 \cdot x, x\}$$



We can construct it like a basic PyTorch module, analogously to the MultiLayer Perceptron which we built (but not trained) at the end of Lab 1.

In [20]:
```python
class LeakyReLU(torch.nn.Module): #activation func defined as Module!!!
    # we might remove it since we don't add anything
    def __init__(self):
        super().__init__()

    def forward(self, data):
        return torch.max(data, data*0.01)
```

and that's it. We may plug it into a neural network module and it'll work just fine, both for the forward and backward pass.

If we want, we can also use it as-is:

In [21]:
```python
leaky_relu = LeakyReLU()

leaky_relu(torch.arange(-10,10)) # is identical to leaky_rely.forward(torch.arange(-10, 10))
```

Out[21]:
```
tensor([-0.1000, -0.0900, -0.0800, -0.0700, -0.0600, -0.0500, -0.0400, -0.0300,
        -0.0200, -0.0100,  0.0000,  1.0000,  2.0000,  3.0000,  4.0000,  5.0000,
         6.0000,  7.0000,  8.0000,  9.0000])
```

let us test its autodiff functionality:

In [22]:
```python
x = torch.tensor([1.0, -1.0], requires_grad=True)

y = leaky_relu(x).sum() # sum to get one single value out of it. (for the automatic differentiation)

print("y:", y)

y.backward()

print("dy/dx:", x.grad)
```

```
y: tensor(0.9900, grad_fn=<SumBackward0>)
dy/dx: tensor([1.0000, 0.0100])
```

We see that the gradient gets calculated automatically without our intervention in defining a gradient function.

But what if that was not already implemented in PyTorch? What if we needed to use some function that cannot be constructed by using PyTorch built-ins?

In this case, we must define a function class which inherits from `torch.autograd.Function`.

An autograd Function inherits from `torch.autograd.Function` class and has two compulsory methods: `forward` and `backward`, whose meaning should be obvious to all.

Both functions have a compulsory first argument which is the **context**, `ctx` for brevity. From the context we can infer informations about the entities involved in the calculation of the gradient.

The context is **built upon calling the `forward` method**, so that, during the `backward` call, we can obtain the info such what tensors have been used in `forward` and whether a tensor requires or not the grad.

In our case, the derivative is the following: $\frac{\partial \text{LeakyReLU}}{\partial x} = \begin{cases} 1 \text{ if } x > 0 \\ 0.01 \text{ if } x \leq 0 \end{cases}$, so we only need to save x, i.e., the data coming into the module.

Moreover, the backward method needs an additional argument, `output_grad`, which conveys information about the gradient which is *entering* the Function (be mindful, we're running *backward*, so a gradient *enters the function* upstream w.r.t. the forward pass).

This is necessary in order to build a cascade of sequential module, each applied after the other. This calls for the application of the **chain rule** for the computation of the gradient of **compositions of functions**:

$$z = g(f(x)) :$$
$$y = f(x) \wedge z = g(y)$$

Then, switching to the derivative:

$$\Rightarrow \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

So, it becomes immediately overt the necessity of having an **incoming** gradient which you use to multiply with the gradient produced by the current module, the result of which gets passed on to the previous node in the computational graph.

In [23]:
```python
class LeakyReLU_Fun(torch.autograd.Function):
    @staticmethod # mind the decorator
    def forward(ctx, input_):
        ctx.save_for_backward(input_) # the parameters that will be involved in the gradient and we can retrive the
        return torch.max(input_, input_ * 0.01)
    @staticmethod
    def backward(ctx, grad_output):
        input_, = ctx.saved_tensors # these are the variables which we need to backpropagate the gradient to (only
        # the gradient is 1 for positive x's, 0.01 for negative x's
        grad_input = torch.ones_like(input_)
        grad_input[input_<0] = 0.01
        # now, we need to rescale for the grad_output
        grad_input *= grad_output
        '''
        a valid alternative (maybe better performing?):
        grad_input = grad_output.clone()
        grad_input[input_<0] *= 0.01
        '''
        return grad_input
```

In [24]:
```
fun = LeakyReLU_Fun.apply #to get an instance of LeakyRelu_Fun
x = torch.linspace(-5,5,11, requires_grad=True)
y = fun(x)
z = y.sum()
z.backward()
```

In [25]:
```
x
```

Out[25]:
```
tensor([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.],
       requires_grad=True)
```

In [26]:
```
x.grad
```

Out[26]:
```
tensor([0.0100, 0.0100, 0.0100, 0.0100, 0.0100, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000, 1.0000])
```

Let us then rivisit our `LeakyReLU` module from before

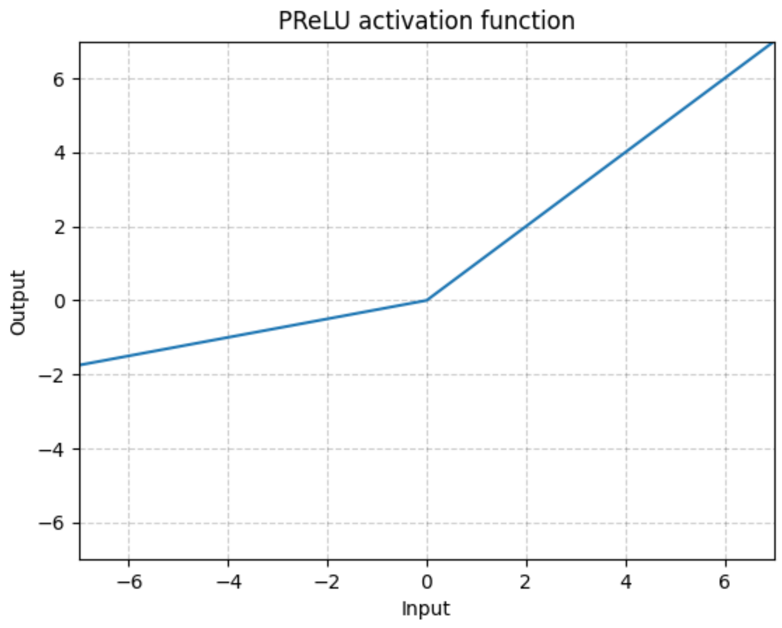
In [27]:
```
class LeakyReLU_Better(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return LeakyReLU_Fun.apply(X)
```

In [28]:
```
LeakyReLU_Better()(x)
```

Out[28]:
```
tensor([-0.0500, -0.0400, -0.0300, -0.0200, -0.0100,  0.0000,  1.0000,  2.0000,
         3.0000,  4.0000,  5.0000], grad_fn=<LeakyReLU_FunBackward>)
```

Building a custom parametric module

We wish to extend our Leaky ReLU module to the Parametric ReLU: $\text{ParamReLU} = \max\{\alpha \cdot x, x\}, \alpha \in [0, 1)$.



Parametric ReLU with $\alpha = 0.25$

In [29]:
```
class ParamReLU_Fun(torch.autograd.Function):
    @staticmethod # mind the decorator
    def forward(ctx, input_, alpha:float):
        assert alpha >= 0 and alpha < 1, f"alpha should be >= 0 and < 1. Found {alpha}."
        ctx.save_for_backward(input_) # the parameters that will be involved in the gradient
        ctx.alpha = alpha # note that we don't use self.alpha: we need to store it!
        return torch.max(input_, input_ * alpha)

    @staticmethod
    def backward(ctx, grad_output):
        input_, = ctx.saved_tensors # these are the variables which we need to backpropagate the gradient to (only
        grad_input = grad_output.clone() # In PT clone is copy
        grad_input[input_<0] *= ctx.alpha
        return grad_input, None
```

In [30]:
```
class ParamReLU(torch.nn.Module):
    def __init__(self, alpha):
        super().__init__()
        self.alpha = alpha

    def forward(self, X):
        return ParamReLU_Fun.apply(X, self.alpha)
```

In [31]:
```
prelu = ParamReLU(0.25)
x = torch.linspace(-5,5,11, requires_grad=True)
y = prelu(x)
z = y.sum()
z.backward()
print(x.grad)
```

```
tensor([0.2500, 0.2500, 0.2500, 0.2500, 0.2500, 1.0000, 1.0000, 1.0000, 1.0000,
        1.0000, 1.0000])
```

We have covered:

1. The construction of a non-parametric differentiable module
2. The construction of a parametric, non-trainable, differentiable module

What's missing?

A construction of a parametric trainable and differentiable model, which we will (hopefully) see once we know how to train models with SGD.

Extra

Backpropagation

Let us suppose we have the following calculation

$\mathbf{x} = [1,\ 2,\ \ -1,\ 3,\ 5]$

$y = f(\mathbf{x}) = \log\{[\exp(x_1 * x_2)]^2 + \sin(x_3 + x_4 + x_5) \cdot x_5\}$

See video in the Teams files for manual backpropagation!

Find

$\nabla f(\mathbf{x})$

In [ ]:
```
# try it in Python!
x_1 = 1
x_2 = 2
x_3 = -1
x_4 = 3
x_5 = 5
```

In [ ]:
```
import math
math.exp(2)**2
```

Backpropagation with PyTorch modules

$\mathbf{x} = [1,\ 2,\ \ -1,\ 3,\ 5]^\top,\ \ \mathbf{w} = [3,\ 0,\ 1,\ \ -3,\ 0.5]^\top$

$y = \sigma(\mathbf{w}^\top \mathbf{x})$, where σ is the sigmoidal function $\frac{1}{1+\exp(-x)}$

In [2]:
```
import torch
class TrivialBackpropagationExample(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = torch.nn.Linear(in_features=5, out_features=1, bias=False)
        self.module1.weight.data = torch.Tensor([3, 0, 1, -3, .5])

    def forward(self, data):
        return torch.sigmoid(self.module1(data))
```

Suppose that the loss function is the binary cross-entropy

$$BCE(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

and that the ground truth is 1.

```
In [5]: def bce_loss(y_hat, y):
            return (-1/len(y)) * (y_hat * y.log() + (1 - y_hat) * (1 - y.log())).sum()

        ground_truth = torch.tensor([1.0])
```

Let us try it...

```
In [7]: model = TrivialBackpropagationExample()

        x = torch.tensor([1,2,-1,3,5], dtype=torch.float32, requires_grad=True)

        y_hat = model(x)
        print(y_hat)

        loss = bce_loss(y_hat, ground_truth)

        loss.backward()

        print(x.grad)
```

```
tensor(0.0110, grad_fn=<SigmoidBackward0>)
tensor([ 0.0326,  0.0000,  0.0109, -0.0326,  0.0054])
```

Homework

1. Complete the Python implementation of the backpropagation exercise in the **Backpropagation** section here above (cell `# try it in Python as homework!`)
   - Create the calculations for obtaining y in PyTorch **using only PyTorch methods and routines**
   - Calculate the gradient
   - Check the values of the gradients and see if it is correct w.r.t. the manual calculations
2. Given the multilayer perceptron defined during the exercises from lab 1:
   - Create 10 random datapoints (with any function you wish, it can be `rand`, `randn` ...) and feed them into the network
   - Given the output, calculate the Cross-Entropy loss with respect to the ground truth $[1, 2, 3, 4, 1, 2, 3, 4, 1, 2]$ (classes from 1 to 4). Cross-Entropy loss:

     $$CE(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{10}\sum_{i=1}^{10} y_i^\top \log(\hat{y}_i)$$

     where y_i is the one-hot encoding of the i-th datapoint. For instance, $y_1 = [1, 0, 0, 0]^\top$. ***Note: there is an extremely handy PyTorch function for getting a one-hot encoding out of a vector, so don't try anything fancy.***
   - Backpropagate the error along the network and inspect the gradient of the parameters connecting the input layer and the first hidden layer.
3. Execute the python script `utils/randomized_backpropagation_formula.py`. This creates a formula $f(\mathbf{x})$ with randomized operators and values. Create the computational graph from this formula, do (by hand) the forward pass, then calculate (by hand) $\nabla f(\mathbf{x})$ using the backward gradient computation. Do the same calculation on PyTorch to check the correctness of your calculations. *Note: The formula created by this script is linked to your name and surname, which you have to input before*. The solution to this exercise *should* be submitted as a scan/good quality picture of a piece of paper (or you can do it on a touch screen and submit the image...), but other formats are acceptable as well.