Laboratory 3 – Pytorch and Deep Networks

Objectives of this lab:

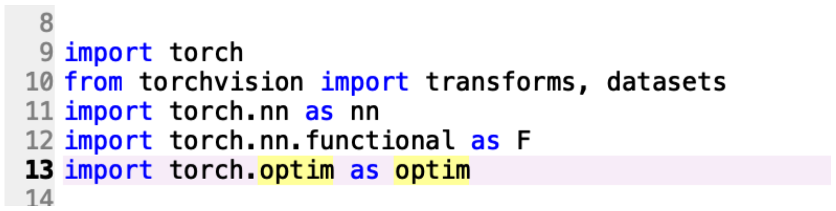
1. *Understand how to use Pytorch to write a Neural Network*
2. *Write a neural network with multiple hidden layers*

In the previous Laboratories we have seen how to write a Neural Network with one hidden layer to solve various problems. The main issue with writing a network from scratches is that it becomes quite tedious to work with multiple layers, especially when, in the training phase, we want to tweak our network. As an example, if we wanted to change the activation functions, the error functions or the optimizer (for instance using something different than classic gradient descent) that would imply rewriting the vast majority of the code.

Luckily there are several libraries in Python that help us automatize the process of writing a network in a modular way. The one we are going to explore in this module is Pytorch (although you are encouraged to check others as well, such as Tensorflow or Keras).

Pytorch should be installed by now, if it's not, please follow the instructions at https://pytorch.org/get-started/locally/. Do not worry about installing CUDA, we will not make use of the GPU in these laboratories (but if you want to know how to use it for your computations, do not hesitate to ask!).

Once that is done you should be able to import the Pytorch libraries. Start by creating a new python file and writing the following:



```python

import torch
from torchvision import transforms, datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

```

If the installation process went smoothly you should be able to save the file and run it without errors. If not, then most likely there was some problem during the installation process, and you might need to go through the previous steps again.

Pytorch uses a specific data structure to handle data, called “tensor”. While some of you might be familiar with the formal notion of tensor (the one from multilinear algebra), in the context of deep learning, a tensor is the generalization of an array or a matrix in an arbitrary number of dimensions (for instance, a vector and a matrix would be a 1-dimensional and a 2-dimensional tensors respectively). The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor.

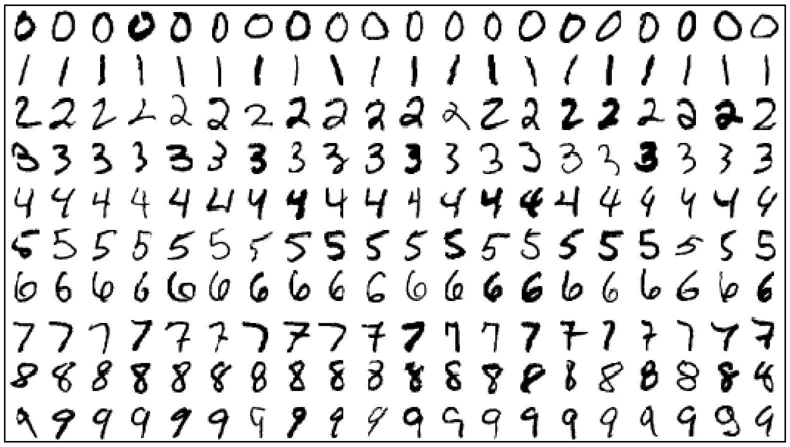
Scalar	Vector	Matrix	Tensor
1	$\begin{bmatrix} 1 \\ a \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ a & b & c \end{bmatrix}$	$\begin{bmatrix} [1\ 2] & [5\ 6] & [7\ 8] \\ [a\ b] & [c\ d] & [e\ f] \end{bmatrix}$

A great feature of Pytorch is “Autograd”. PyTorch tensors keep track of the operations that involved them and they automatically provide the series of derivatives with respect to their inputs. This means you won’t need to write expressions for the backpropagation algorithm and compute derivatives by hand, as we did in the previous laboratories. Pytorch, no matter how complicated your model is, will compute them automatically.

Once that is done, let’s see how to build a neural network using a classical dataset as a benchmark.

The dataset we are going to use is called MNIST and it contains a series of images of handwritten digits. The goal is to teach our network to properly classify the digits.

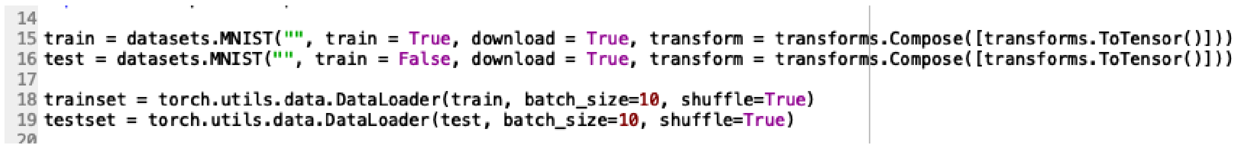
Here is an example of the images that we are going to use:



The database is organized as follows:

- The input is a 28 by 28 matrix (2-dimensional tensor), which corresponds to the pixels of the image.
- The output is an integer between 0 and 9.

In order to download the database on your computer please add the following lines to your code (do not forget to import torch and datasets):

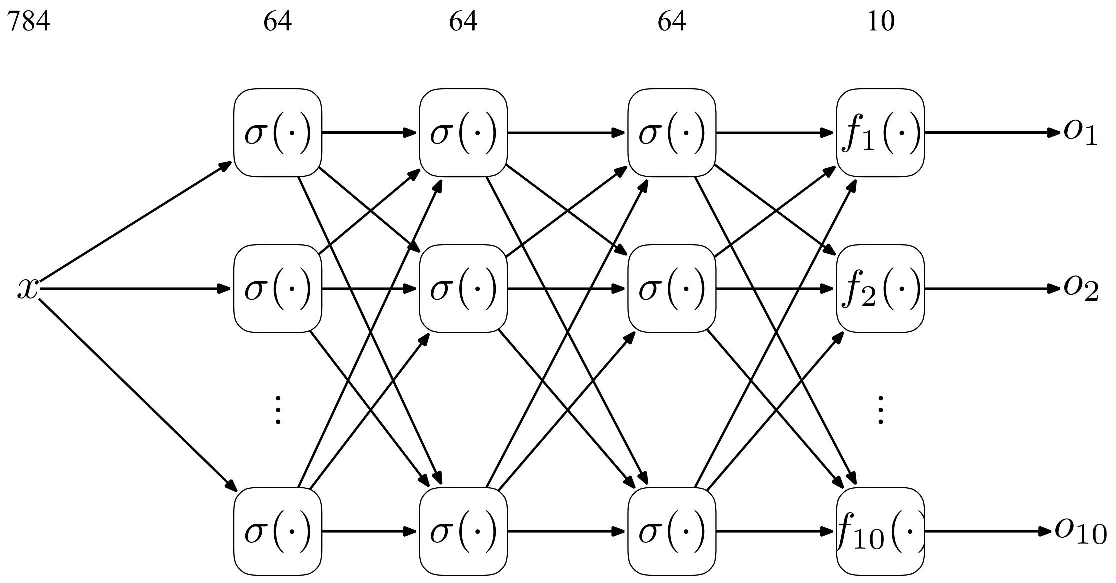


```
train = datasets.MNIST("", train = True, download = True, transform = transforms.Compose([transforms.ToTensor()]))
test = datasets.MNIST("", train = False, download = True, transform = transforms.Compose([transforms.ToTensor()]))

trainset = torch.utils.data.DataLoader(train, batch_size=10, shuffle=True)
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=True)
```

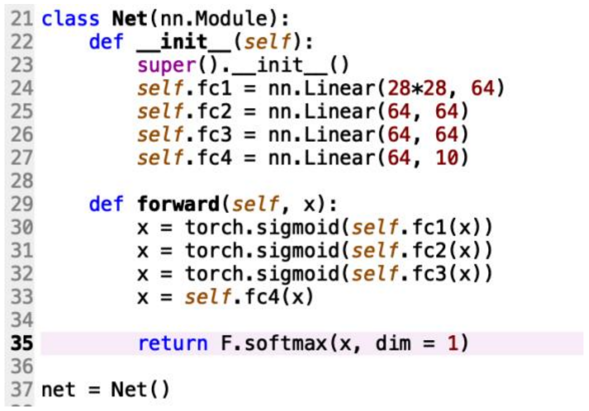
Do not worry about the meaning of the code above, we are using it just to get our data, which is going to be stored in the variables “trainset” and “testset”.

Now, suppose that we want to use the following architecture in order to properly classify our dataset:



According to the picture above, we want a network that takes as an input a 784 (28*28 pixels) dimensional vector (or a 784 1-dimensional tensor), have 3 hidden layers of 64 neurons each and an output layer of 10 units (the integers from 0 to 9). The activation functions should be sigmoid for the hidden layers and the softmax for the output layer.

This is a really big network compared to the ones we used before and it would be quite complex to write it from scratches. Luckily, the code for a network such as this one is extremely simple to write in Pytorch:



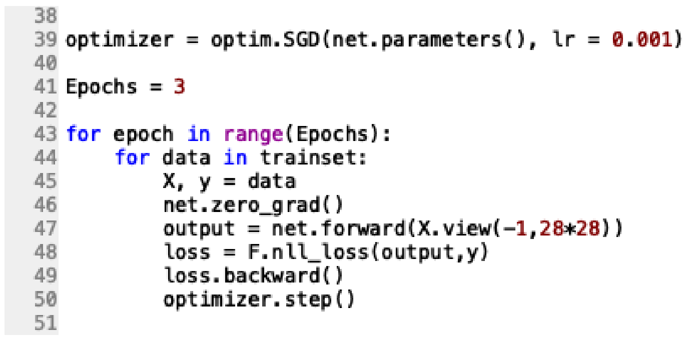
```
21 class Net(nn.Module):
22     def __init__(self):
23         super().__init__()
24         self.fc1 = nn.Linear(28*28, 64)
25         self.fc2 = nn.Linear(64, 64)
26         self.fc3 = nn.Linear(64, 64)
27         self.fc4 = nn.Linear(64, 10)
28 
29     def forward(self, x):
30         x = torch.sigmoid(self.fc1(x))
31         x = torch.sigmoid(self.fc2(x))
32         x = torch.sigmoid(self.fc3(x))
33         x = self.fc4(x)
34 
35         return F.softmax(x, dim = 1)
36 
37 net = Net()
```

To begin, you will need to create a class from nn.Module. A PyTorch module is a Python class derived from the nn.Module base class. This module contains the building blocks to create neural networks. The architecture is defined in the __init__ method: there are various functions that you can use to create different layers. The function nn.Linear is the basic structure of a layer where an affine transformation is applied to the input (i.e., $w^T x + b$) and it is the one you have been using in the past laboratories. In the next laboratory we will see how to apply a convolutional transformation.

For the time being let's focus on nn.Linear: this function receives two arguments (the dimension of the input and the dimension of the output) and initializes a certain amount of weights (tensors) accordingly. For example, the code in line 24 says that we are creating a layer of 64 neurons with 784 inputs each (that means approximately 50176 weights in the first layer alone). Accordingly, the second and the third hidden layer have 64 inputs and 64 outputs (these numbers need to match each other!), while the output layer has 64 inputs and 10 outputs.

The forward method is the one we will use to compute the output of our network to an input x. The method torch.sigmoid computes the sigmoid function of a given input (lines 30 to lines 32) and the method F.sofmax computes the softmax function of a given input. Finally, in line 37 we create an instance of the neural network.

In order to finish our first network on Pytorch we still need to setup the optimizer, the error function, to compute the gradient with respect to the loss and then train for a number of iterations. All of this can be summarized in the following lines of code:

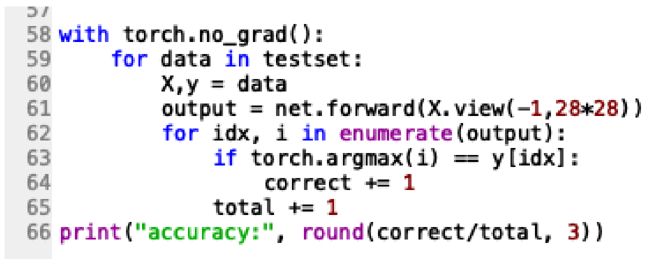


```
38
39 optimizer = optim.SGD(net.parameters(), lr = 0.001)
40
41 Epochs = 3
42
43 for epoch in range(Epochs):
44     for data in trainset:
45         X, y = data
46         net.zero_grad()
47         output = net.forward(X.view(-1,28*28))
48         loss = F.nll_loss(output,y)
49         loss.backward()
50         optimizer.step()
51
```

That's quite an improvement with respect to the code that we had to write for the much simpler networks before! Let's take a look at each line individually:

- Line 39: This sets up the optimization method you are going to use in the backpropagation algorithm. In this example we are using a stochastic gradient descent;
- Line 41: We are setting up the number of training epochs;
- Line 42: We iterate over the training data (represented by the variable trainset);
- Line 45: We assign input (X) and labels (y);
- Line 46: This sets up the gradient stored in each variable of our network to zero. If we didn't use this command, then at each iteration autograd will keep adding the new value of the gradient to the previous one;
- Line 47: We compute the output using the forward method. The method .view transforms our 2 dimensional tensor input (a 28 by 28 matrix) to a 1 dimensional one (a 784 vector).
- Line 48: This sets up the loss function. Since we are working with a classifier, we use a cross entropy loss function;
- Line 49: This line computes the gradient with respect to the loss function over each parameter of the network (again, without the command at line 46, this gradient would accumulate over each iteration leading to an incorrect training);
- Line 50: Finally, this line updates the parameters of our network according to the optimization algorithm and the gradient stored within each variable.

Now proceed to train the network and use the following code to test if the training was successful:



```
with torch.no_grad():
    for data in testset:
        X,y = data
        output = net.forward(X.view(-1,28*28))
        for idx, i in enumerate(output):
            if torch.argmax(i) == y[idx]:
                correct += 1
            total += 1
print("accuracy:", round(correct/total, 3))
```

The “torch.no_grad()” simply means that your network won’t update the gradient stored in each variable during the test session (as there is no loss to be minimized!).

At this point, run the code and see what happens.

This shouldn’t work (the main problem here is that we are not training for enough epochs and that stochastic gradient descent is generally very slow). Now try to play with the network a bit. This website https://pytorch.org/docs/stable/index.html provides with the full documentation of Pytorch and allows you to change your network in any way you see fit. In particular, try the following:

- Change the sigmoid function with a ReLu function (F.relu instead of torch.sigmoid)
- Change the optimizer from stochastic gradient descent to Adam (optim.Adam instead of optim.SGD)
- Try and modify the number of layers and the number of neurons in the hidden layer (be careful that the numbers are consistent between contiguous layers)

Can you see a significant change in the performance of the network? Why do you think that’s the case?