Laboratory 4 – Pytorch and Convolutional Deep Networks

Objectives of this lab:

1. *Write a convolutional neural network for the MNIST Database*

In this laboratory, we are going to learn how to use Pytorch to write Convolutional Neural Networks. We are still going to use the MNIST database and the code to load it, as in the previous lab.

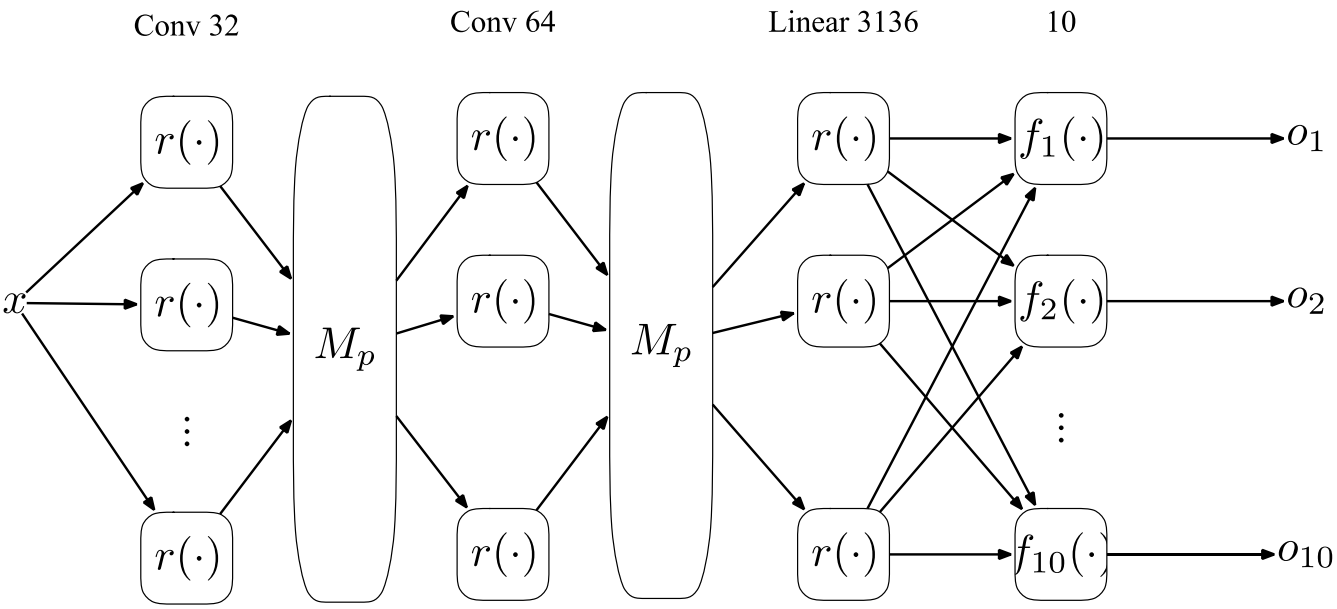
```
import torch
from torchvision import transforms, datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
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)
```

Unlike the previous lab though, we are going to make use of convolutional layers, instead of linear ones to classify the inputs.

Our network is going to look like this



Where the the activation function for each layer is the rectified linear function (ReLu) (r), the pooling layer M_p takes the maximum value of a patch of the image and the output layer is made of softmax functions.

Make a copy of the code we used for the previous laboratory and let's change the Net class.

```
20
21 class Net(nn.Module):
22     def __init__(self):
23         super().__init__()
24         self.conv1 = nn.Conv2d(1, 32, 5, padding=2)
25         self.conv2 = nn.Conv2d(32, 64, 5, padding=2)
26
27
28         self.fc1 = nn.Linear(64*7*7, 128)
29         self.fc2 = nn.Linear(128, 10)
30
31     def convs(self, x):
32
33         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
34         x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
35
36         return x
37
38     def forward(self, x):
39         x = self.convs(x)
40         x = x.view(-1, 64*7*7)
41
42         x = F.relu(self.fc1(x))
43         x = self.fc2(x)
44         return F.softmax(x, dim=1)
45
```

The function nn.Conv2d is the convolutional layer that we are interested in. The first two inputs represent the dimensions of the layer, whereas the third one represents the dimension of the convolution kernel (we are using a 5x5 kernel in this example). The last parameter sets the padding for the image (why did we set it to 2?). The only other new function is the F.max_pool2d, which simply applies the max function over a patch of the image (in this example, we set it to a 2x2 patch).

As you can see, the linear layers are set up in such a way to take the "flattened" output of our convolutional layer, as their input is a vector. Can you explain why the input is set to 64*7*7?

And that's it! All that is left to do now is to slightly modify our previous code (as we are giving the network a matrix as an input this time, instead of a vector) and make it run.

```
net = Net()

optimizer = optim.Adam(net.parameters(), lr = 0.001)

Epochs = 3

for epoch in range(Epochs):
    for data in trainset:
        X, y = data
        net.zero_grad()
        output = net.forward(X)
        loss = F.nll_loss(output,y)
        loss.backward()
        optimizer.step()

    print(loss)


correct = 0
total = 0

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

In this code we are using again Adam as the optimizer, as that's the standard for neural networks at the time these notes have been written and you should have realized by now how poorly the classical gradient descent performs in these kinds of tasks.

You'll notice that this code takes significantly more time to run than the previous one. Do you know why that's the case?

Now we encourage you to try to make the same modifications that you made in the previous lab (modifying the layers, changing the number of neurons, changing optimizer and activation functions) and, in addition to that, modify the pooling layer (e.g., using the average function instead of the max function), modify the dimension of the convolution kernels and to modify the dimension of the padding (be careful to the inputs of the various layers!). Does the performance of the network improve? Is it reasonable to expect to reach a 100% classification accuracy?