CS 454

Introduction to Machine Learning and Artificial Neural Networks

Prof. Dr. Ethem Alpaydın

Fall 2021

Homework #4

Neural Networks (PyTorch)

by

Abdullah Saydemir

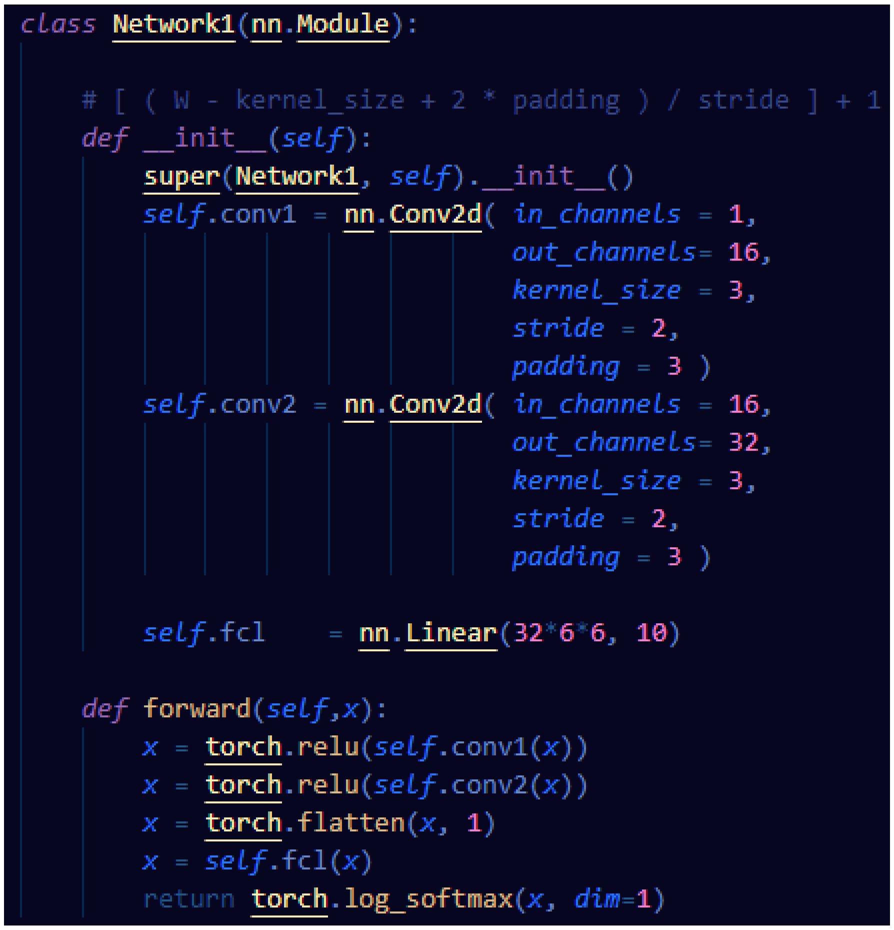
S014646

December 22, 2021

Since I do not own the code, I will push it to the branch and later delete it after grades are announced. Code and log files are available via this link after the deadline.

Part 1 - Defining Networks

I defined three networks as follows. Variables in the convolutional layers will be changed.



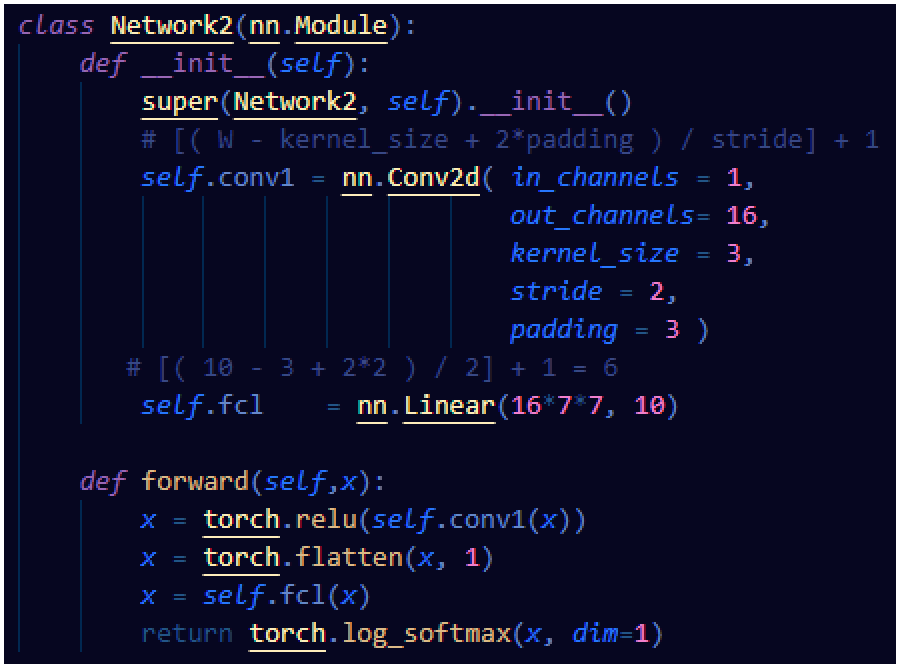
```
class Network1(nn.Module):

    # [ ( W - kernel_size + 2 * padding ) / stride ] + 1
    def __init__(self):
        super(Network1, self).__init__()
        self.conv1 = nn.Conv2d( in_channels = 1,
                                out_channels= 16,
                                kernel_size = 3,
                                stride = 2,
                                padding = 3 )
        self.conv2 = nn.Conv2d( in_channels = 16,
                                out_channels= 32,
                                kernel_size = 3,
                                stride = 2,
                                padding = 3 )

        self.fcl    = nn.Linear(32*6*6, 10)

    def forward(self,x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.flatten(x, 1)
        x = self.fcl(x)
        return torch.log_softmax(x, dim=1)
```

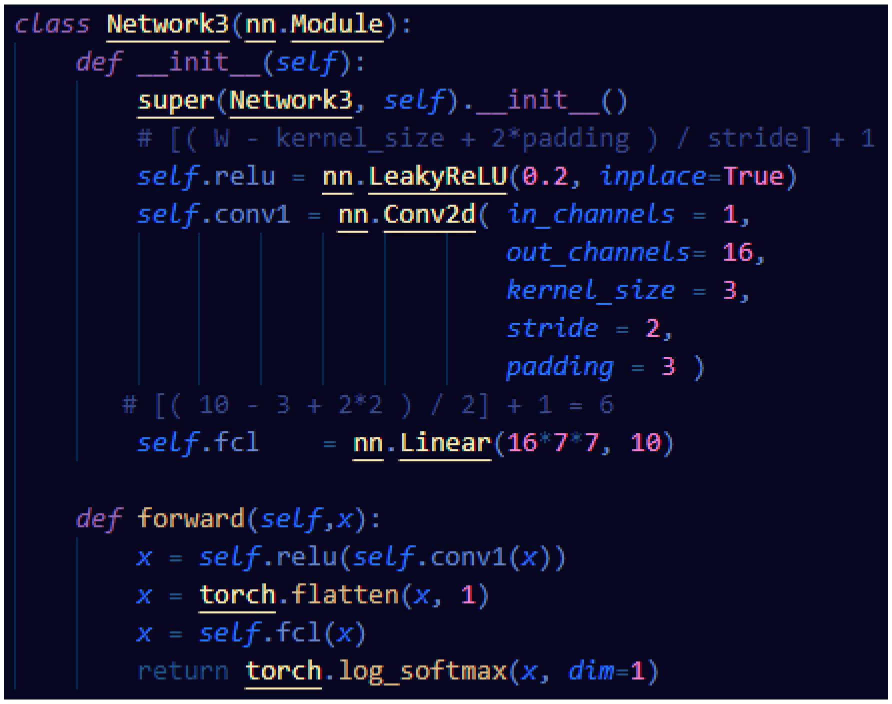
Network1 has two fully connected layers followed by a fully connected layer. Activation function is ReLU instead of sigmoid.



```python
class Network2(nn.Module):
    def __init__(self):
        super(Network2, self).__init__()
        # [( W - kernel_size + 2*padding ) / stride] + 1
        self.conv1 = nn.Conv2d( in_channels = 1,
                                out_channels= 16,
                                kernel_size = 3,
                                stride = 2,
                                padding = 3 )
       # [( 10 - 3 + 2*2 ) / 2] + 1 = 6
        self.fcl     = nn.Linear(16*7*7, 10)

    def forward(self,x):
        x = torch.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = self.fcl(x)
        return torch.log_softmax(x, dim=1)
```

Network2 has 1 convolutional layer and 1 fully connected layer. Activation function is also ReLU.



```python
class Network3(nn.Module):
    def __init__(self):
        super(Network3, self).__init__()
        # [( W - kernel_size + 2*padding ) / stride] + 1
        self.relu = nn.LeakyReLU(0.2, inplace=True)
        self.conv1 = nn.Conv2d( in_channels = 1,
                                out_channels= 16,
                                kernel_size = 3,
                                stride = 2,
                                padding = 3 )
       # [( 10 - 3 + 2*2 ) / 2] + 1 = 6
        self.fcl     = nn.Linear(16*7*7, 10)

    def forward(self,x):
        x = self.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = self.fcl(x)
        return torch.log_softmax(x, dim=1)
```

Network3 is simply Network2 with leaky ReLU.

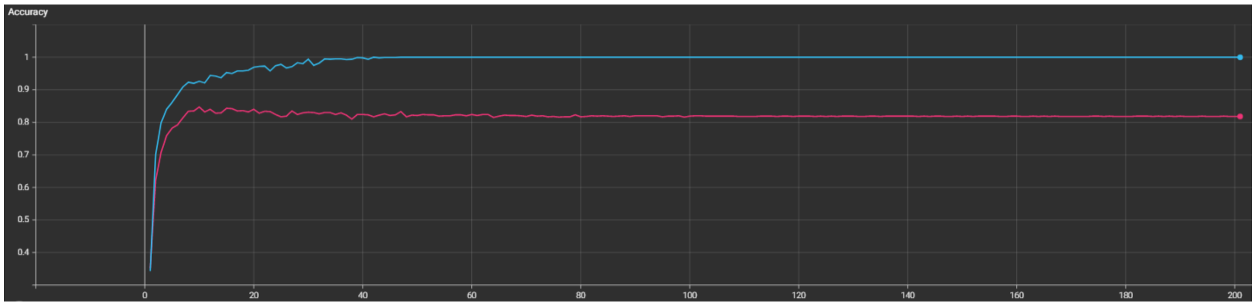
Part 2 - Trying Different Parameters & Plotting Accuracy

There needs to be 3 graphs total, one for each network. I would like to experiment more to compare each variable's effect to the accuracy. I will provide 3 graphs for each section and compare their best results later.

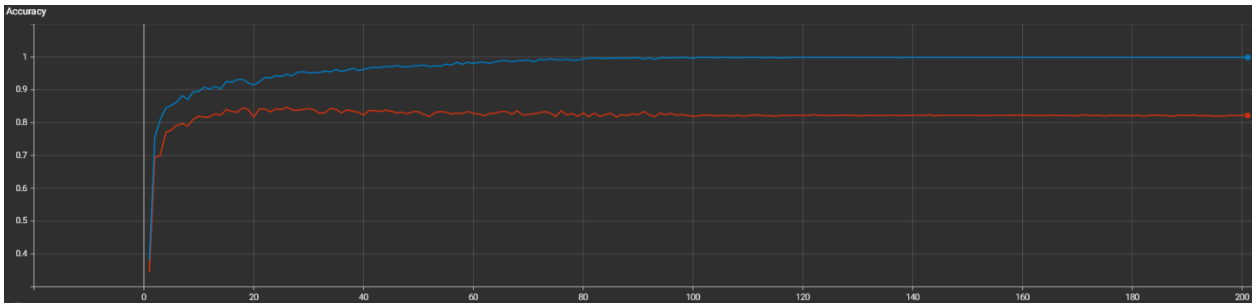
a) Kernel Size

kernel_size = 1 (no padding, stride = 1)

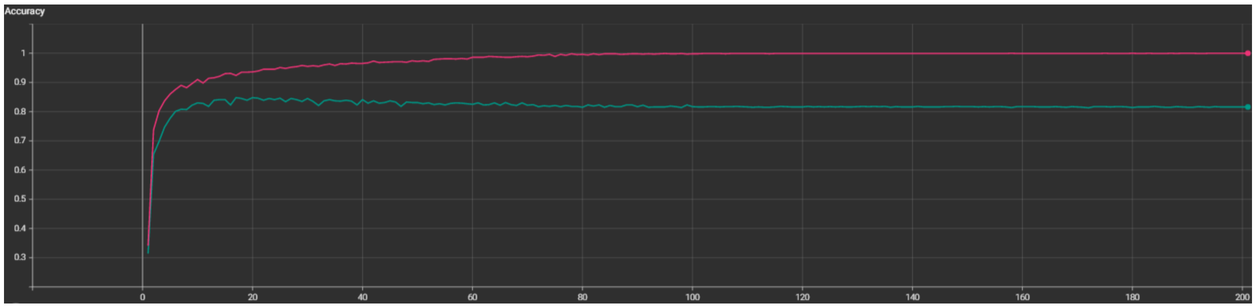
Network1 :



Network2 :



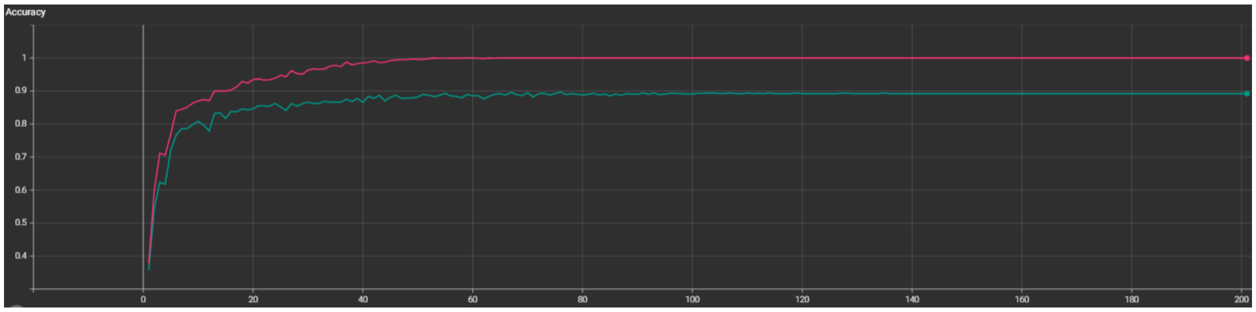
Network3 :



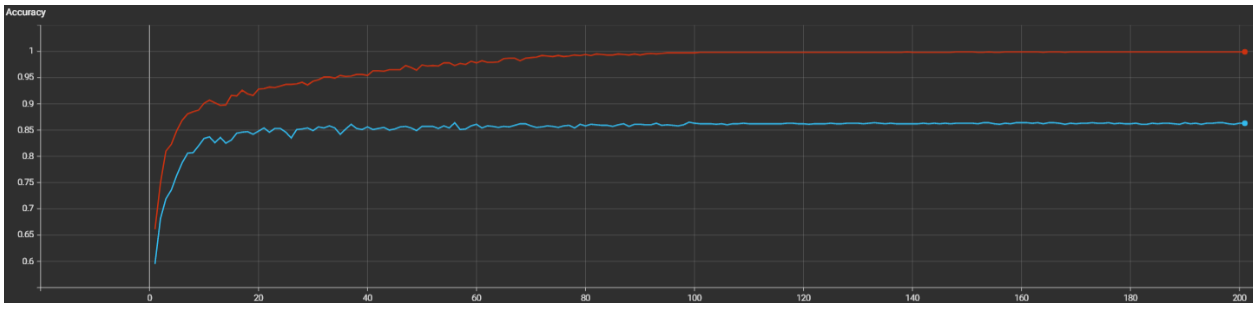
Network1 easily memorized the training data and provided good results for testing data. Network2 was so close to memorizing but stuck at 0.999. That means the second layer only contributed 0.001 :) Leaky ReLU was also able to convert, though it took 190 epochs to memorize.

kernel_size = 4 (no padding, stride = 1)

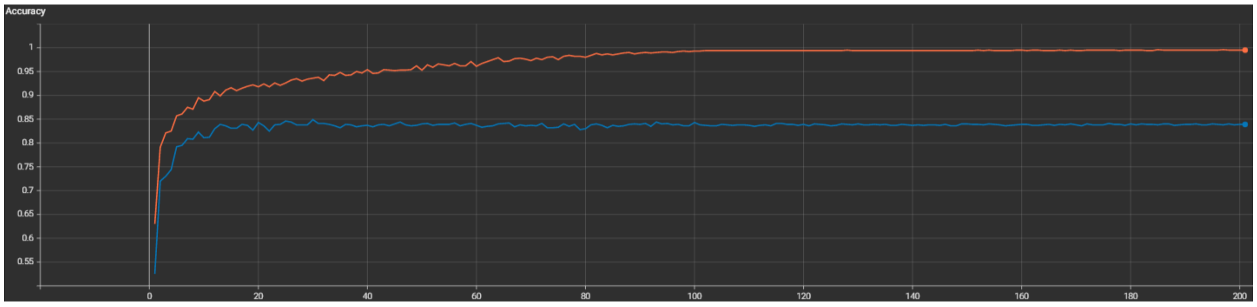
Network1 :



Network2 :



Network3 :

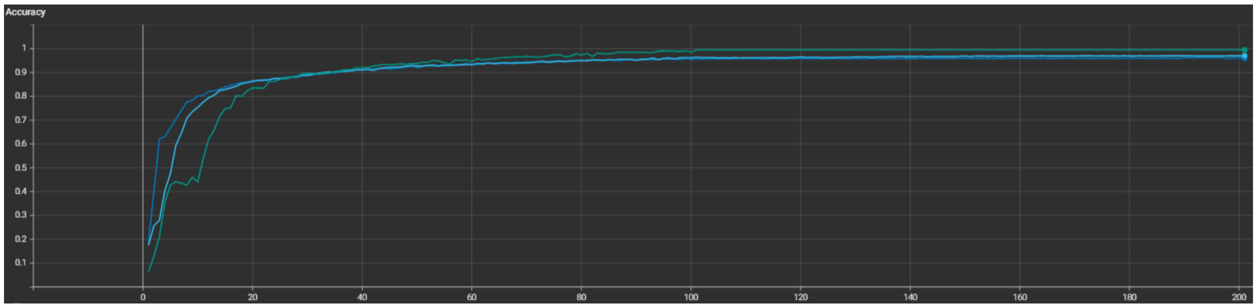


From the first two experiments, Network1 seems the most powerful one, because it really gets the better accuracy in 70-80 epochs. However, the other two networks give good results in 70-80 epochs as well. Network2 is very very close to memorizing the training data. This time, Network3 cannot memorize as well. Maybe a negative slope of 0.2 is too much.

I think providing 2 graphs for training and testing is better for comparison. Therefore, I switched to this representation.

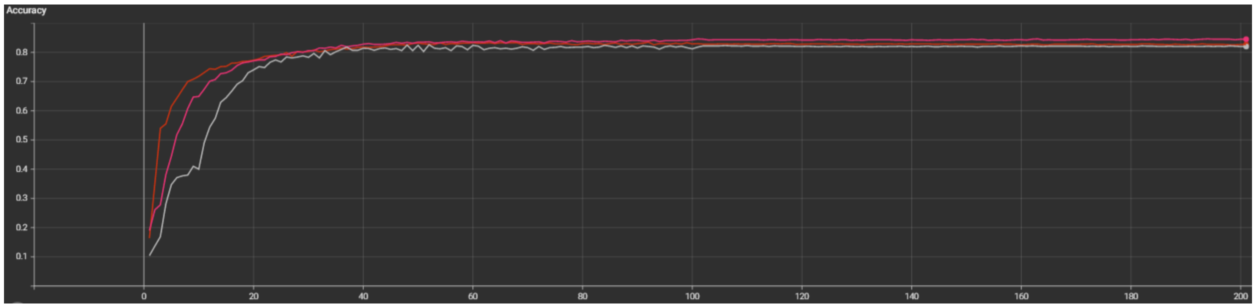
kernel_size = 10 (1 for second layer) (no padding, stride = 1)

Training :



Network1 > Network2 >= Network3

Testing :



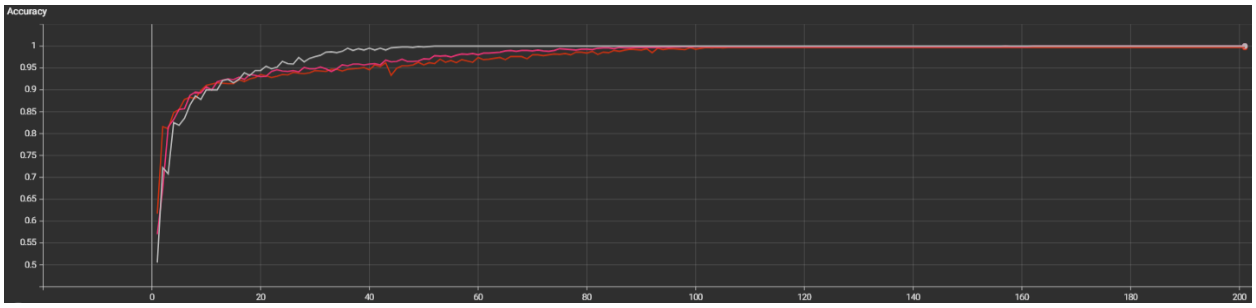
Network2 > Network3 >= Network1

For the first time Network1 could not memorize the training data and testing accuracy was lower. Also, there is a bump in the beginning of the chart. Network2 was following Network1 and Network3 was the last for training. On the other hand, Leaky ReLU gave better results in testing data and Network1 was the poorest.

b) Stride

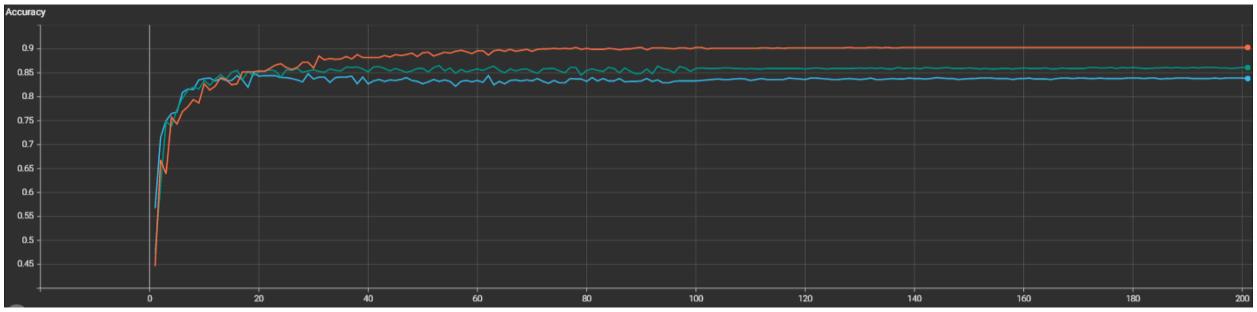
stride = 1 (no padding, kernel_size = 3)

Training :



Network1 >= Network2 >= Network3

Testing :

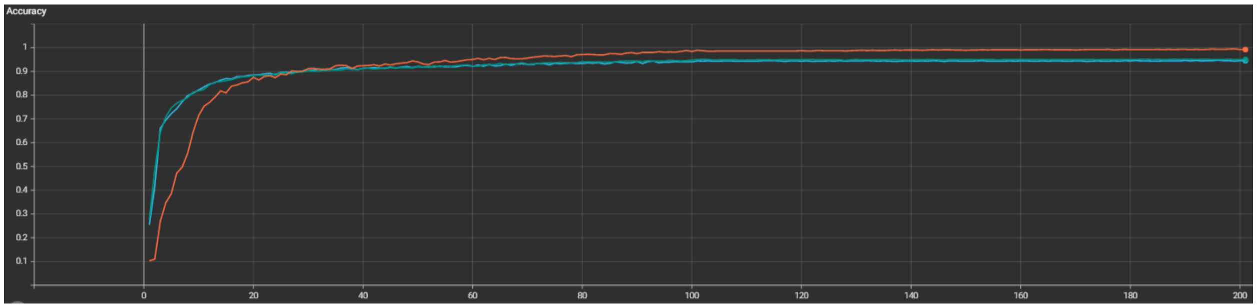


Network1 > Network2 > Network3

Network2 was able to memorize the data; however, Network1 outperformed Network2 by 5% on testing data. Leaky ReLU provided poorer accuracy in both data.

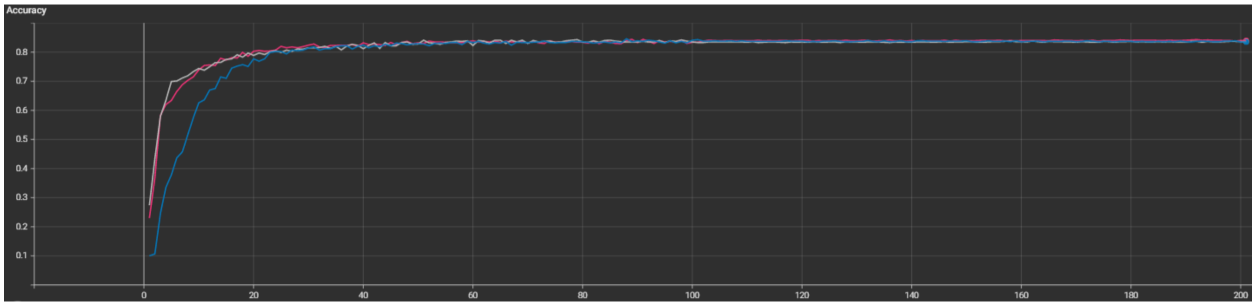
stride = 3 (no padding, kernel_size = 3)

Training :



Network1 > Network2 > Network3

Testing :

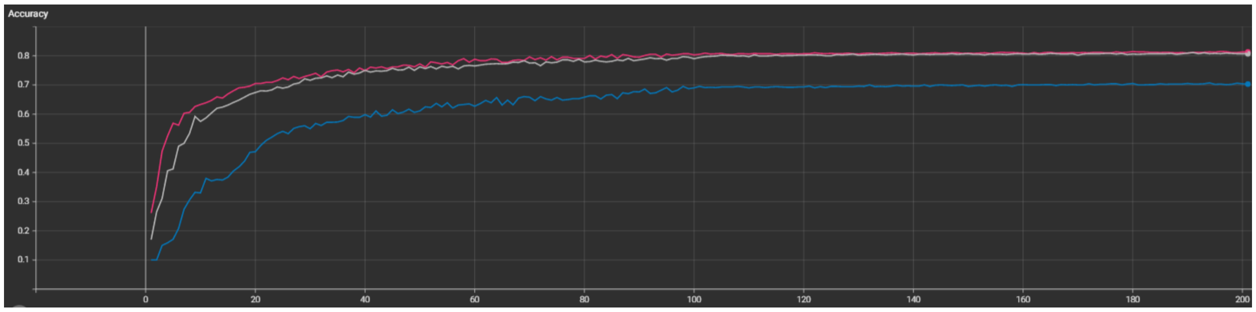


Network1 > Network3 > Network2

I was expecting lower accuracy compared to the previous experiment and it was as I expected. Because kernel_size = stride we do not sample the same pixels again. This kind of hurts learning. It is interesting that Network1 was the best but it started slow in the beginning.

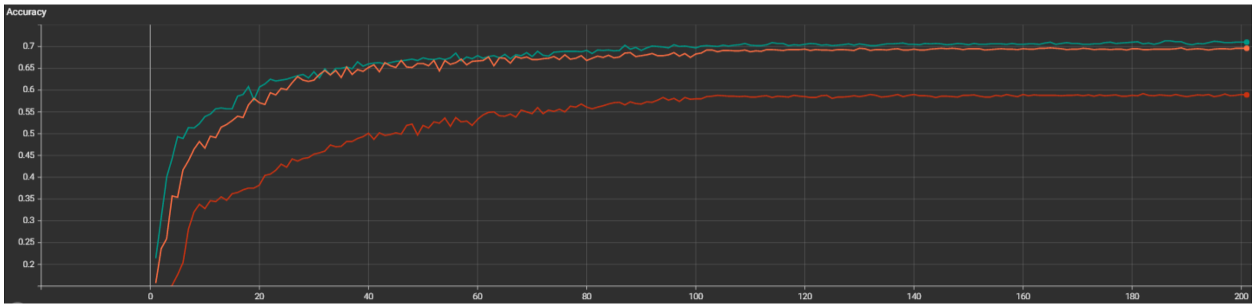
stride = 4 (no padding, kernel_size = 2) Expectation : Very low accuracy compared to previous ones, since we do not take some pixels into account.

Training :



Network2 > Network3 > Network1

Testing :



Network2 > Network3 > Network1

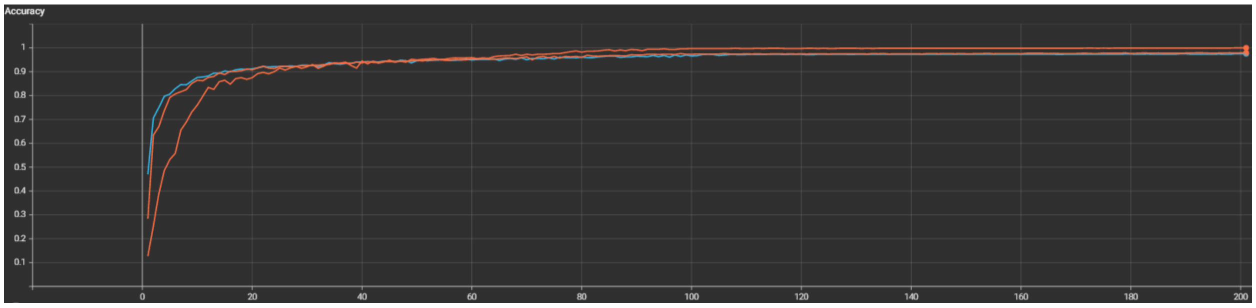
Accuracy decreased as expected. We have some pixels never used in the learning process. Network1 is far worse if it does not have enough samples.

c) Padding

I will not try 0 padding since all of the experiments above have no padding.

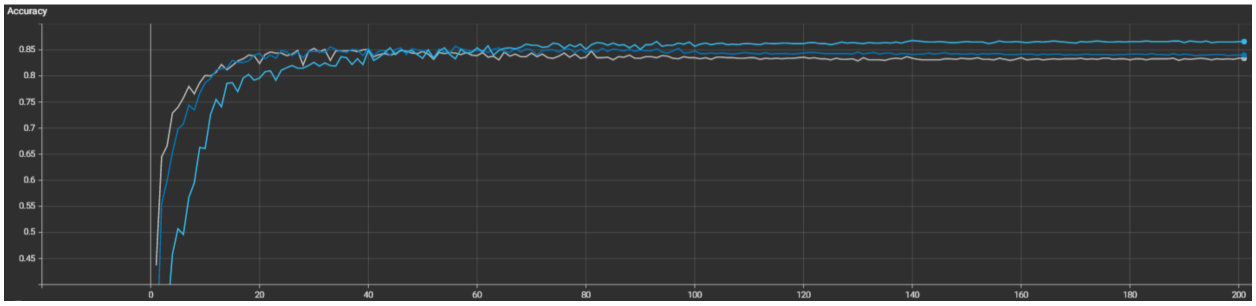
padding = 1 (stride = 2, kernel_size = 3) Expectation : I think padding does not have too much effect on accuracy. I am expecting a high accuracy since other variables are normal.

Training :



Network1 > Network2 > Network3

Testing :

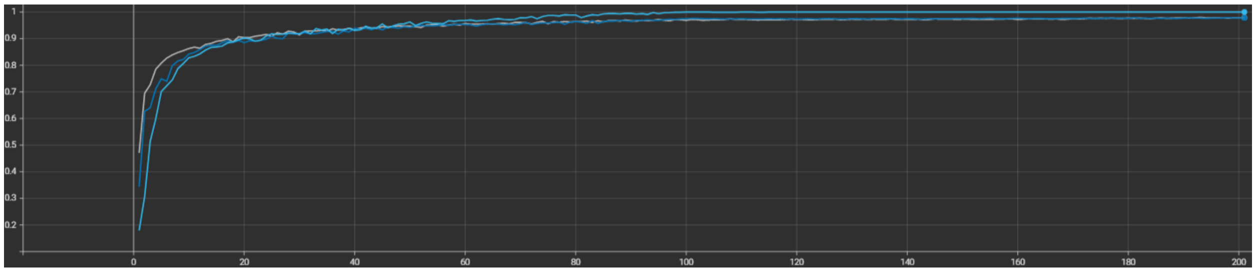


Network1 > Network2 > Network3

Nothing unusual here. I don't know why there are two orange lines in one graph… Sorry for that…

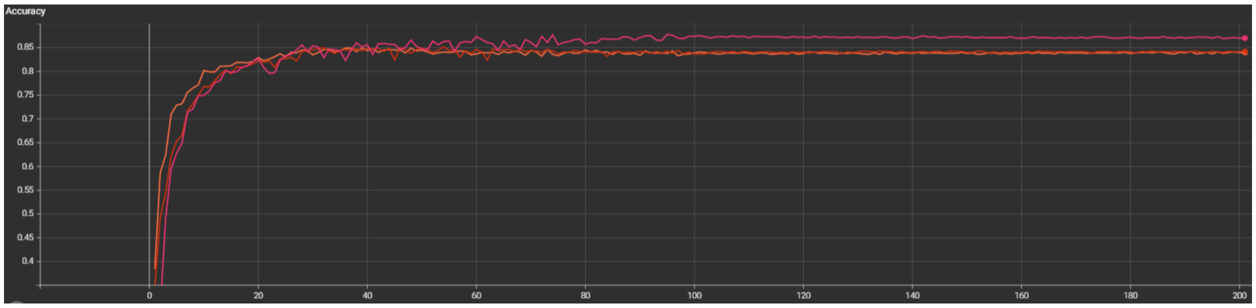
padding = 2 (stride = 2, kernel_size = 3) Expectation : I think padding will affect accuracy. I am expecting lower accuracy compared to the previous experiment.

Training :



Network1 > Network3 >= Network2

Testing :

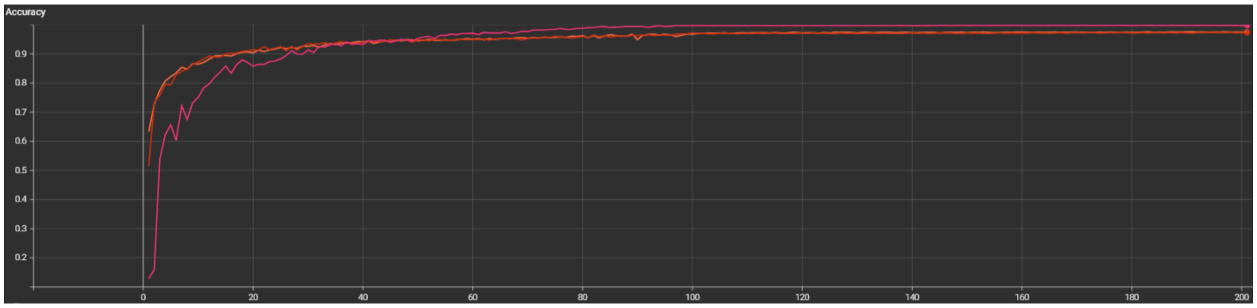


Network1 > Network2 >= Network3

Yes. Accuracy is lower than before. However, it is not really significant. It is not even differentiable from the graph.

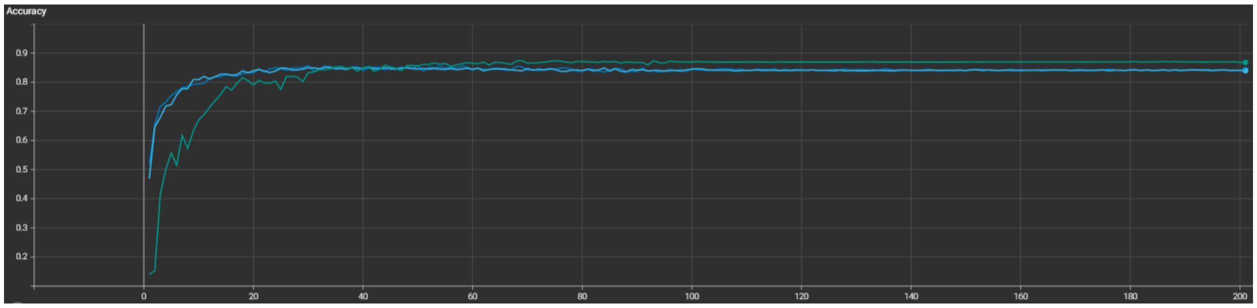
padding = 3 (stride = 2, kernel_size = 3)

Training :



Network1 > Network2 >= Network3

Testing :



Network1 > Network2 >= Network3

I did not have time to change in-out channels. I could have also used different activation functions.

Comparison of Best Accuracies of Three Networks

For Network1, the best result was observed when there was no padding, stride was 1 and kernel_size was 3 or 4. It provided 90% accuracy.

For Network2, the best result was observed when there was no padding, stride was 1 and kernel_size was 4. Accuracy is around %86.

For Network3, the best result was observed when there was no padding, stride was 1 and kernel_size was 4. Accuracy is around %85.

Observations

Most probably, given parameters were not perfect for the first two layers. By optimizing those, higher accuracy can be achieved in test data. However, to my observations and according to my experiments, using complex network structures does not really affect the accuracy for reasonable parameters. Again, most probably this is a false claim, since we did not even touch some of the network variables (batch size, learning rate, gamma etc.)

Ceteris paribus, changing one variable is not a good idea for experimenting. Rather, changing the ratios among the variables would be better. For example, stride may significantly affect the accuracy if kernel size is similar to stride size (kernel size 3, stride 3) but it may help achieve better results if kernel size is higher (e.g. kernel size 5, stride 2).

At the end of the notebook, there was a weights variable that I assume shows a similar image to digits that we obtained in HW3. I could not find a way to show or print it.

Worth mentioning that Network2 and Network3 performed better than Network3 in poor conditions i.e. when the parameters were absurd.

What I learned

- I learned how to define a network in the PyTorch library. I am not really comfortable with changing the system but I can at least adjust the parameters.
- I learned what is a stride, padding, kernel_size and how changing those parameters affect the accuracy.
- I am familiar with jupyter notebooks and I can do basic operations on it. Still a lot to learn about the extensions…
- It may not be the case that most complex networks give better results.

Questions that appeared in my mind

- Is there any formula or algorithm that gives the best variable size if other variables are provided. For example, what is the best kernel size if the image is 28x28, padding is 2 and stride is 2? Is there any method other than trial and error?
- We did not transform the data too much while importing it to the torch. If we had groups of different features in the data, how could I get specific features from the tensors? Is it available as it is in pandas like df[‘label’]?