Neural Networks for Images (67103): Exercise 1

By: Amit Halbreich, ID: 208917393

Theoretical Questions – Answers

1. By Discrete Delta Function definition:

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & otherwise \end{cases}$$

The response of the LTI System to the input signal is called the impulse response h[n]. We got:

$$h[n] = L\big[\delta[n]\big]$$

where L is a given LTI System.

Consider a signal denoted by x[n], since any continuous time signal can be expressed as a sum of scaled and shifted unit impulses functions, by splitting the signal to impulse train we get:

$$x[n] = \sum_{k=-\infty}^{\infty} \delta[n-k]x[k]$$

For a linear function L, we get by considering the linearity of the signal, and by the shifting property of the unit impulse function, the response of LTI System to the input signal x[n], By linearity we get:

$$y[n] = L[x[n]] = L[\sum_{k=-\infty}^{\infty} \delta[n-k]x[k]] = \sum_{k=-\infty}^{\infty} L\big[\delta[n-k]\big]x[k] = \sum_{k=-\infty}^{\infty} h[n-k]x[k]$$

By the Time Invariance property, we get $L\big[\delta[n-k]\big] = h[n-k]$. For this reason, we can substitute between the two sides of the equation and get:

$$y[n] = \sum_{k=-\infty}^{\infty} h[n-k]x[k]$$

And finally, by the convolution definition we have:

$$y[n] = (x * h)[n]$$

Consider now time-shifting compute $L[x[n+m]]$ as:

$$\boldsymbol{L[x[n+m]](j)} = L\{\sum_{k=-\infty}^{\infty} x[k]\delta[n+m-k]\}(j) = \sum_{k=-\infty}^{\infty} x[k]L\big[\delta[n+m-k]\big](j) =$$

$$= \sum_{k=-\infty}^{\infty} x[k]L\big[\delta[n+m-k]\big](j) = \sum_{k=-\infty}^{\infty} x[k]h[j-(k-m)] =$$

$$= \sum_{k=-\infty}^{\infty} x[k]h[(j+m)-k] = \sum_{k=-\infty}^{\infty} x[k]h[j-k] = \mathbf{L[x[n]](j+m)}$$

Where we used the trait of shift-invariance of the LTI operator **L**:

$$L\big[x[n]\big] = L[x[n+m]]$$

Comparing this to L[x[n]], we see that the condition L[x[n+m]] = L[x[n]] is equivalent to $h[j-(k-m)] = h[j-k]$ which means that the impulse response h[j] is shift-invariant, and hence, L is a convolution operator. The input signal that will output the underlying filter of the convolution is a delta function, i.e., $L\big[\delta[n]\big] = h[n]$.

In conclusion, we have proved that any Discrete Linear Time Invariant system is derived directly from its impulse response h[n] and moreover, the system response y[n] = L[x[n]] which is a linear operator performed on a signal x[n] equals to the result of the convolution:

$$y[n] = (x * h)[n]$$

Therefore, the given condition corresponds to a convolution, and the filter h[k] is the response of the linear operator L to a unit impulse δ[k].

In other words, the input signal that will output the underlying filter of the convolution is a unit impulse signal δ[k]. By applying the linear operator L to this input signal, we obtain the filter h[k] as the output.

2. The order in which the 2D activation map is reshaped and fed into an FC layer can be important. One common approach is to reshape the activation map into a 1D vector by flattening it row by row, column by column or even diagonally. A second approach may be to reshape it randomly or in an order that will not preserve the spatial relations between pixels in the same row/column or diagonal.
   For example, if the original 2D map has dimensions (height, width), one could reshape it into a 1D tensor of shape (height*width) by concatenating the rows or by concatenating the columns.

   For instance, if the original 2D map represents an image, reshaping by rows will preserve the spatial relations between pixels in the same row, whereas reshaping by columns will preserve the spatial relations between pixels in the same column. But if we will reshape the 2D activation map of the image in random ordering or in a way that will not preserve spatial relations between the image's pixels and then feeding it into an FC Layer may cause a decrease in the network's accuracy and it will be more lossy and less likely to classify the image to its specific class correctly.
   In conclusion, we will strive to preserve spatial relations between the image's pixels using flattening to 1D vector according to rows/columns and will

choose the method that will produce better results - that is considered as "Good" ordering. On the other hand, ordering randomly the 2D activation map to 1D vector and then feeding it into an FC Layer is a "Bad" ordering method.

3. a. **The ReLU activation function is not LTI.**

As we can infer from the function's definition, it uses max operator which is generally non-linear:

$$f(x) = \max(0, x) \qquad (1)$$

where x = an input value

According to equation 1, the output of ReLu is the maximum value between zero and the input value. An output is equal to zero when the input value is negative and the input value when the input is positive. Thus, we can rewrite equation 1 as follows:

$$f(x) = \begin{cases} 0, if\ x < 0 \\ x, if\ x \geq 0 \end{cases} \qquad (2)$$

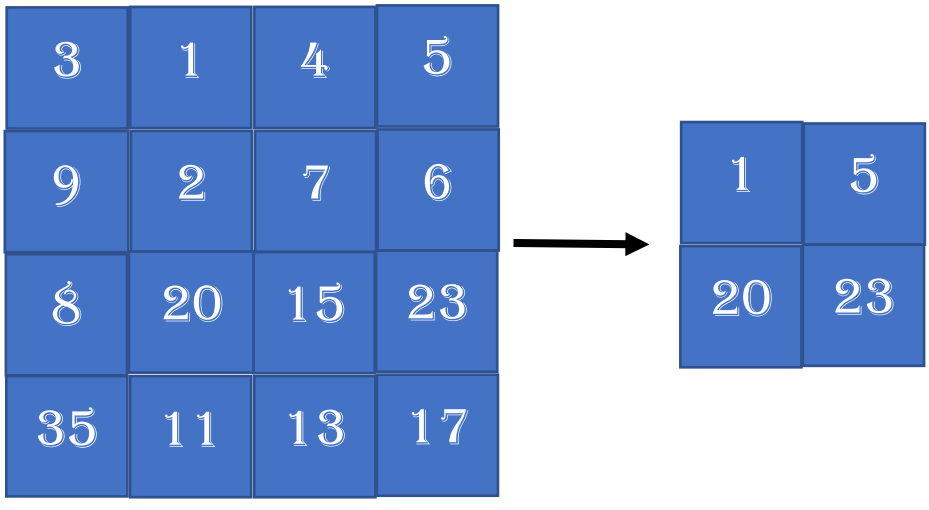
where x = an input value

This is because it is a nonlinear function that transforms the input data in a way that is not proportional to the input. Specifically, the ReLU function sets all negative values to zero, while leaving positive values unchanged. This introduces a nonlinearity to the model, which means that the output is not directly proportional to the input. As a result, the **ReLU function violates the linearity property of LTI systems and therefore isn't LTI.**

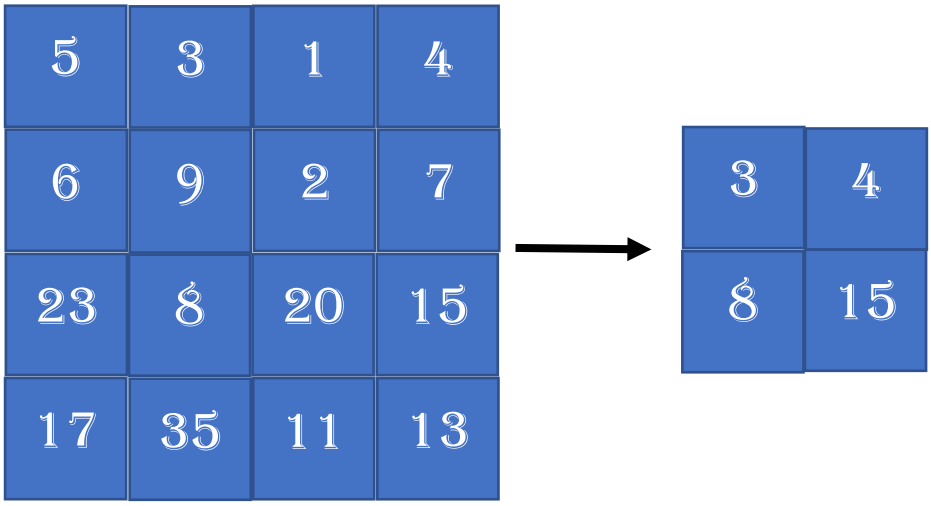
b. **The Strided Pooling Layer is not LTI.** This is because it operates by selecting a single pixel (typically the top-right pixel) from each block of pixels and is not shift invariant. This operation is not translation-invariant because if we shift the blocks of pixels in the input data different pixels will be chosen as the output.

Counter Example:

Consider an image **I** and apply a strided pooling operator of size 2x2 choosing the top-right pixel used on it:



Now shifting all the pixels 1 step to the right by column:



We can easily detect that we got a different output layer as a result of the shifting.

Therefore, the strided pooling operation suggested is not LTI.

c. **The addition of a Bias is not LTI.** This is because the bias term violates the linearity trait of the LTI System.
The homogeneity trait is violated when introducing the bias offset vector since for every image **I** and operator **B**: **B(I) = I + b** denoted as bias operator we have:

$$\lambda I + b = \boldsymbol{B}(\boldsymbol{\lambda I}) \neq \boldsymbol{\lambda}\big(\boldsymbol{B}(\boldsymbol{I})\big) = \lambda(I + b) = \lambda I + bI$$

This means that the **bias addition is therefore not LTI.**

d. **The multiplication with a fully-connected matrix is not LTI.** This is because the output of the multiplication operation is not translation-invariant and therefore isn't LTI. If we shift the input by some amount, the output will be shifted by a different amount. Similarly, if we scale the input, the output will be scaled by a different factor. To refute the LTI trait I will give a counter example.

Counter example:
Let's denote by **I** an input image and by **W** a fully-connected matrix. The output image **O** is given by the matrix product **(1) O = WI**. Now, if we shift the input signal by a vector a, the output signal becomes **(2) O' = W(I + a)**. Expanding this expression, we get **(3) O' = WI + Wa**.
So, the output is shifted by a different vector **Wa**, which is not the same as the input shift a. This means that the fully-connected multiplication operation is not translation-invariant.

CODING QUESTIONS – ANSWERS

1.

GRAPHS:

Underfitting

$$Learned\ Paremeters = 69706$$

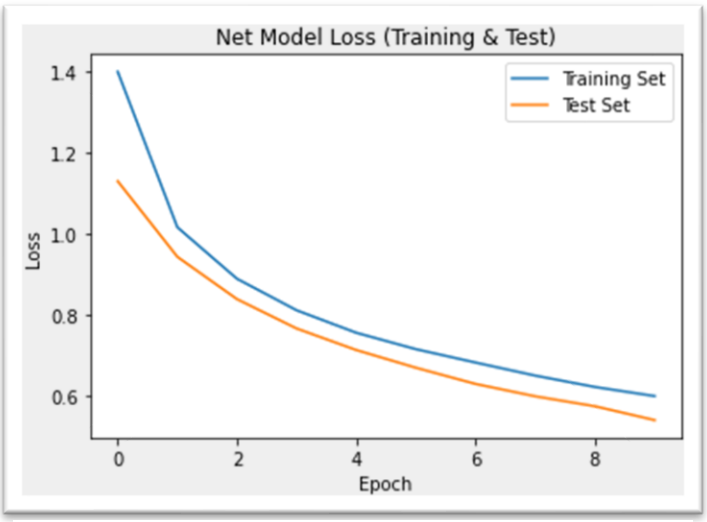
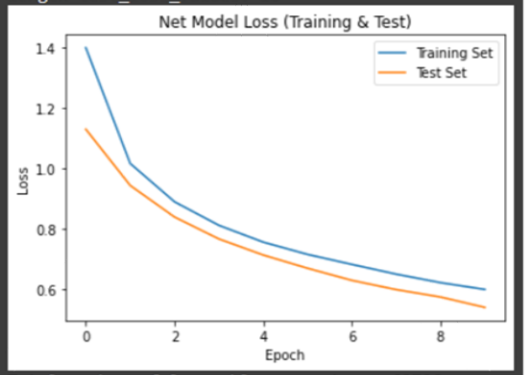


Figure 1: Underfitting Q1

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64 * 5 * 5, 10)
        # self.fc2 = nn.Linear(120, 84)
        # self.fc3 = nn.Linear(84, 10)
        # self.fc2 = nn.Linear(120, 84)
        # self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        # x = F.relu(self.fc1(x))
        # x = F.relu(self.fc2(x))
        # x = self.fc3(x)
        x = (self.fc1(x))
        return x


net = Net()
net.to(device)
```

Figure 2: Network Details

```
Epoch 1: Train loss: 1.398, Test loss: 1.129
Epoch 2: Train loss: 1.015, Test loss: 0.942
Epoch 3: Train loss: 0.888, Test loss: 0.838
Epoch 4: Train loss: 0.810, Test loss: 0.765
Epoch 5: Train loss: 0.755, Test loss: 0.712
Epoch 6: Train loss: 0.714, Test loss: 0.668
Epoch 7: Train loss: 0.681, Test loss: 0.628
Epoch 8: Train loss: 0.649, Test loss: 0.597
Epoch 9: Train loss: 0.620, Test loss: 0.573
Epoch 10: Train loss: 0.598, Test loss: 0.539
length loss_values: 10
length test_loss_values: 10
```




```
Total number of learnable parameters: 69706
Finished Training
```

Figure 3: Learned Parameters & Raw Loss Data

Overfitting

$$Learned\ Paremeters = 106008$$

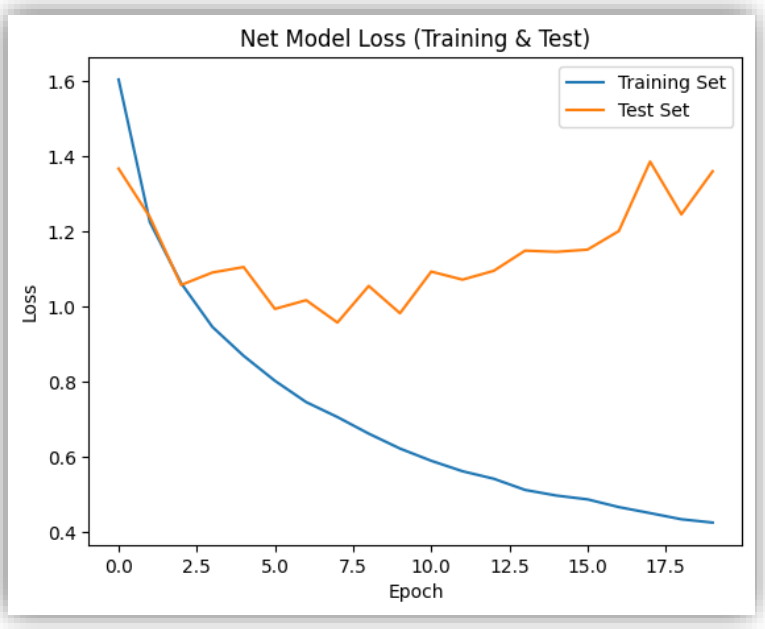


Figure 1: Overfitting Q1

```
Epoch 1: Train loss: 1.605, Test loss: 1.368
Epoch 2: Train loss: 1.225, Test loss: 1.239
Epoch 3: Train loss: 1.063, Test loss: 1.058
Epoch 4: Train loss: 0.946, Test loss: 1.091
Epoch 5: Train loss: 0.869, Test loss: 1.106
Epoch 6: Train loss: 0.803, Test loss: 0.994
Epoch 7: Train loss: 0.746, Test loss: 1.017
Epoch 8: Train loss: 0.706, Test loss: 0.958
Epoch 9: Train loss: 0.662, Test loss: 1.055
Epoch 10: Train loss: 0.622, Test loss: 0.983
Epoch 11: Train loss: 0.590, Test loss: 1.094
Epoch 12: Train loss: 0.562, Test loss: 1.072
Epoch 13: Train loss: 0.542, Test loss: 1.096
Epoch 14: Train loss: 0.512, Test loss: 1.149
Epoch 15: Train loss: 0.497, Test loss: 1.146
Epoch 16: Train loss: 0.487, Test loss: 1.152
Epoch 17: Train loss: 0.467, Test loss: 1.201
Epoch 18: Train loss: 0.451, Test loss: 1.386
Epoch 19: Train loss: 0.434, Test loss: 1.246
Epoch 20: Train loss: 0.425, Test loss: 1.361
length loss_values: 20
length test_loss_values: 20
```


```
Total number of learnable parameters: 106008
Finished Training
```

Figure 3: Learned Parameters & Raw Loss Data

```
[ ]  import torch.nn as nn
     import torch.nn.functional as F


     class Net(nn.Module):
         def __init__(self):
             super().__init__()
             self.conv1 = nn.Conv2d(3, 32, 5)
             self.pool = nn.MaxPool2d(2, 2)
             self.conv2 = nn.Conv2d(32, 16, 5)
             self.fc1 = nn.Linear(16 * 5 * 5, 200)
             self.fc2 = nn.Linear(200, 50)
             self.fc3 = nn.Linear(50, 10)

         def forward(self, x):
             x = self.pool(F.relu(self.conv1(x)))
             x = self.pool(F.relu(self.conv2(x)))
             x = torch.flatten(x, 1) # flatten all dimensions except batch
             x = F.relu(self.fc1(x))
             x = F.relu(self.fc2(x))
             x = self.fc3(x)
             return x


     net = Net()
     net.to(device)
```

Figure 2: Network Details

$$Learned\ Paremeters = 167530$$

Figure 1: Overfitting Q1

Figure 4: Q1 Overall performance Underfitting to Overfitting

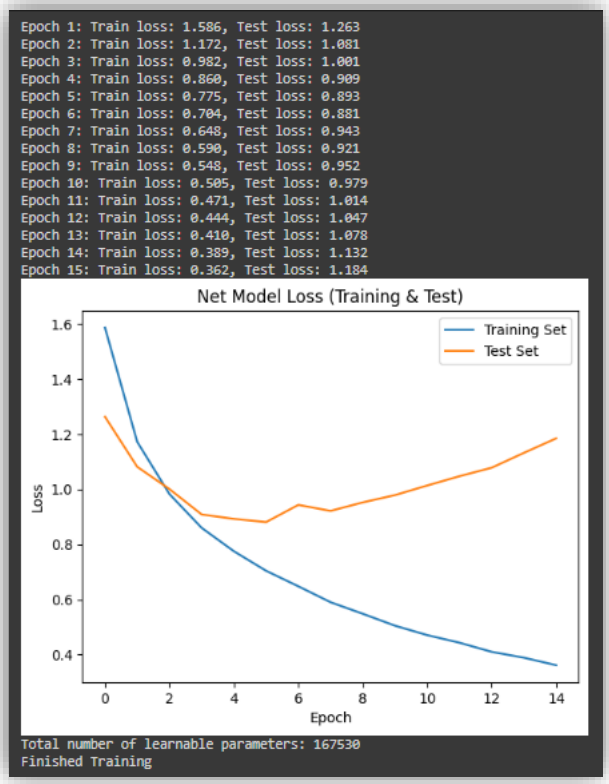


Figure 3: Learned Parameters & Raw Loss Data

```python
# Question 1 - Overfitting
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(64, 32, 5)
        self.fc1 = nn.Linear(32 * 5 * 5, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
net.to(device)
```

Figure 4: Network Details

2. Once we remove all non-linear components from the neural network the network's accuracy decreases drastically and the Train Loss & Test Loss remain pretty high – around **Loss = ~ 1.75**. If we compare it to the efficiency and accuracy of the network we created in Question 1 in which we achieved higher accuracy and lower Loss =~ 0.5, the linear components network is more lossy and less accurate.
   The reason for that is that if we apply only linear operators the network is more restricted to finding more complex patterns which are non-linear when training on the given dataset – patterns such as curves, edges, different abstract shapes are less likely to be identified by a linear component network.
   The components in the given network which are non-linear are:
   ReLU Operator – is non-linear (explained in the theoretical questions above).
   Max Pooling - is non-linear because taking the maximum value of each pooling region is a non-linear function of the input values.

GRAPHS:

Figure 1: Q2 Graph

```
Epoch 1: Train loss: 1.855, Test loss: 1.769
Epoch 2: Train loss: 1.803, Test loss: 1.776
Epoch 3: Train loss: 1.794, Test loss: 1.781
Epoch 4: Train loss: 1.788, Test loss: 1.782
Epoch 5: Train loss: 1.785, Test loss: 1.799
Epoch 6: Train loss: 1.781, Test loss: 1.767
Epoch 7: Train loss: 1.781, Test loss: 1.799
Epoch 8: Train loss: 1.776, Test loss: 1.798
Epoch 9: Train loss: 1.776, Test loss: 1.751
Epoch 10: Train loss: 1.774, Test loss: 1.792
```


```
Total number of learnable parameters: 6434
Finished Training
```

Figure 3: Learned Parameters & Raw Loss Data

```python
# Question 2 - No Linear Components Network
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.avgPool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 8, 5)
        self.fc1 = nn.Linear(8 * 5 * 5, 10)

    def forward(self, x):
        x = self.avgPool(self.conv1(x))
        x = self.avgPool(self.conv2(x))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc1(x)
        return x


net = Net()
net.to(device)
```

Figure 2: Q2 Network Details

$$Learned\ Parameters = 6434$$

3.

GRAPHS:

Using Global Average Pooling version – Using convolution with 5x5 kernel and then Narrowing the network to a single 1x1 with total of 16 channels and run FC Layer that narrows to 10 Neurons at the output:

$$Learned\ Paremeters = 13034$$

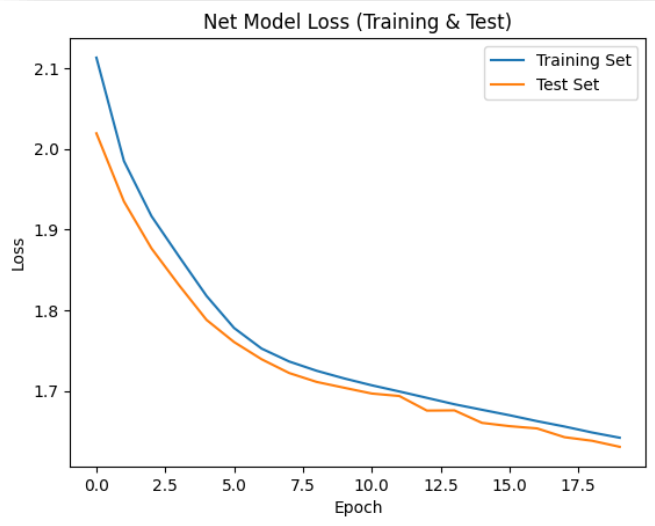
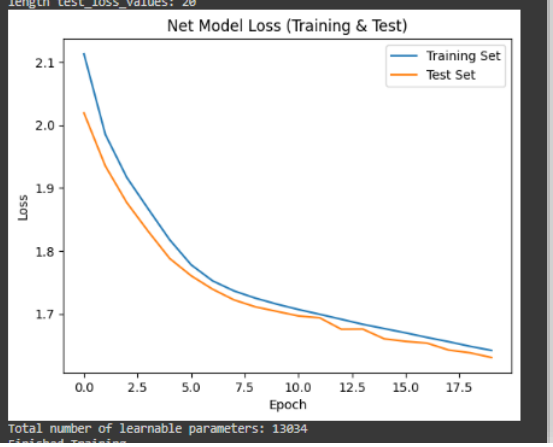


Figure 6: Underfitting Q3

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.globalAvgPool = nn.AdaptiveAvgPool2d((1, 1))
        self.avgPool = nn.AvgPool2d(28,28)
        self.fc1 = nn.Linear(16, 10)
        # self.fc2 = nn.Linear(120, 84)
        # self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.globalAvgPool(F.relu(self.conv1(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc1(x)
        # x = F.relu(self.fc1(x))
        # x = F.relu(self.fc2(x))
        # x = self.fc3(x)
        return x
```

Figure 5: Network Details

```
Epoch 1: Train loss: 2.113, Test loss: 2.019
Epoch 2: Train loss: 1.985, Test loss: 1.935
Epoch 3: Train loss: 1.917, Test loss: 1.877
Epoch 4: Train loss: 1.867, Test loss: 1.831
Epoch 5: Train loss: 1.818, Test loss: 1.788
Epoch 6: Train loss: 1.778, Test loss: 1.761
Epoch 7: Train loss: 1.752, Test loss: 1.739
Epoch 8: Train loss: 1.736, Test loss: 1.722
Epoch 9: Train loss: 1.725, Test loss: 1.711
Epoch 10: Train loss: 1.715, Test loss: 1.704
Epoch 11: Train loss: 1.707, Test loss: 1.697
Epoch 12: Train loss: 1.699, Test loss: 1.694
Epoch 13: Train loss: 1.691, Test loss: 1.676
Epoch 14: Train loss: 1.683, Test loss: 1.676
Epoch 15: Train loss: 1.677, Test loss: 1.660
Epoch 16: Train loss: 1.670, Test loss: 1.656
Epoch 17: Train loss: 1.663, Test loss: 1.654
Epoch 18: Train loss: 1.656, Test loss: 1.643
Epoch 19: Train loss: 1.648, Test loss: 1.638
Epoch 20: Train loss: 1.642, Test loss: 1.631
length loss_values: 20
length test_loss_values: 20
```




```
Total number of learnable parameters: 13034
```

Figure 4: Learned Parameters & Raw Loss Data

Using one convolution layer and then applying a FC layer instantly:

$$Learned\ Paremeters = 54046$$

Figure 8: Overfitting Q3

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 9, 9)
        self.fc1 = nn.Linear(9 * 24 * 24, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc1(x)
        return x
net = Net()
net.to(device)
```

Figure 7: Network Details

```
Epoch 1: Train loss: 1.737, Test loss: 1.689
Epoch 2: Train loss: 1.578, Test loss: 2.349
Epoch 3: Train loss: 1.500, Test loss: 1.583
Epoch 4: Train loss: 1.450, Test loss: 1.746
Epoch 5: Train loss: 1.406, Test loss: 1.643
Epoch 6: Train loss: 1.375, Test loss: 1.888
Epoch 7: Train loss: 1.347, Test loss: 1.752
Epoch 8: Train loss: 1.319, Test loss: 1.714
Epoch 9: Train loss: 1.286, Test loss: 1.848
Epoch 10: Train loss: 1.269, Test loss: 1.903
Epoch 11: Train loss: 1.245, Test loss: 1.885
Epoch 12: Train loss: 1.231, Test loss: 1.895
Epoch 13: Train loss: 1.217, Test loss: 2.606
Epoch 14: Train loss: 1.201, Test loss: 1.938
Epoch 15: Train loss: 1.190, Test loss: 2.107
length loss_values: 15
length test_loss_values: 15
```


```
Total number of learnable parameters: 54846
Finished Training
```

Figure 9: Learned Parameters & Raw Loss Data

The network in Question 1 is a standard CNN architecture that uses convolutional layers and pooling to reduce the spatial dimensions of the input image while increasing the number of feature maps. This architecture has been proven to work well for image classification tasks.

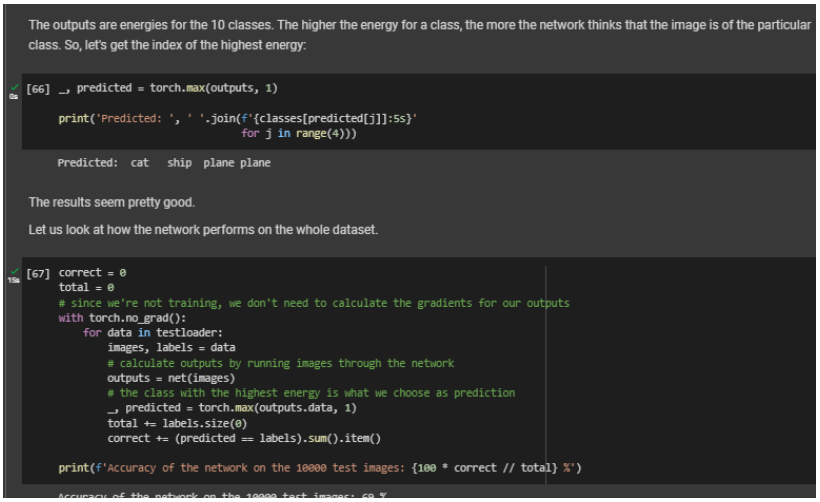
On the other hand, the network in Question 3 is a shallower network that directly maps the output of the first convolutional layer to the output layer, either through an FC layer or global average pooling. This approach may not be as effective in capturing the complex features of an image and may result in lower performance compared to the standard CNN architecture used in Question 1. That is because once we map the output of the first convolution layer to the final output layer, we may have a bigger receptive field, but we get a shallow network by collapsing the original images to 1x1 channels using Global Average Pooling/or connecting the first convolutional layer to FC Layer very quickly without learning regional key features in the given images and we "lose" precision and the network fails to learn important details in the image, in contrary to the network in Question 1 whereas we break down the network to smaller parts by using convolutional layers and pooling to reduce the spatial dimensions of the input image.

It seems that the network in Question 1 is more likely to perform better due to its more complex architecture and proven effectiveness in image classification tasks.

I chose to try both suggestions, the (i) Global Average Pooling and (ii) connecting the first convolutional layer to FC Layer both didn't seem to produce better results than the Network I used in Question 1 and that is due to the reasons I mentioned above.

The highest accuracy reached was 69% and it was with a network presented in Q1:

```
[68] # prepare to count predictions for each class
     correct_pred = {classname: 0 for classname in classes}
     total_pred = {classname: 0 for classname in classes}

     # again no gradients needed
     with torch.no_grad():
         for data in testloader:
             images, labels = data
             outputs = net(images)
             _, predictions = torch.max(outputs, 1)
             # collect the correct predictions for each class
             for label, prediction in zip(labels, predictions):
                 if label == prediction:
                     correct_pred[classes[label]] += 1
                 total_pred[classes[label]] += 1


     # print accuracy for each class
     for classname, correct_count in correct_pred.items():
         accuracy = 100 * float(correct_count) / total_pred[classname]
         print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

Accuracy for class: plane is 77.7 %
Accuracy for class: car   is 85.4 %
Accuracy for class: bird  is 53.9 %
Accuracy for class: cat   is 51.2 %
Accuracy for class: deer  is 71.5 %
Accuracy for class: dog   is 56.2 %
Accuracy for class: frog  is 79.4 %
Accuracy for class: horse is 70.2 %
Accuracy for class: ship  is 77.8 %
Accuracy for class: truck is 75.6 %
```



The outputs are energies for the 10 classes. The higher the energy for a class, the more the network thinks that the image is of the particular class. So, let's get the index of the highest energy:

```
[66] _, predicted = torch.max(outputs, 1)

     print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                                    for j in range(4)))

     Predicted:  cat   ship  plane plane
```

The results seem pretty good.

Let us look at how the network performs on the whole dataset.

```
[67] correct = 0
     total = 0
     # since we're not training, we don't need to calculate the gradients for our outputs
     with torch.no_grad():
         for data in testloader:
             images, labels = data
             # calculate outputs by running images through the network
             outputs = net(images)
             # the class with the highest energy is what we choose as prediction
             _, predicted = torch.max(outputs.data, 1)
             total += labels.size(0)
             correct += (predicted == labels).sum().item()

     print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```