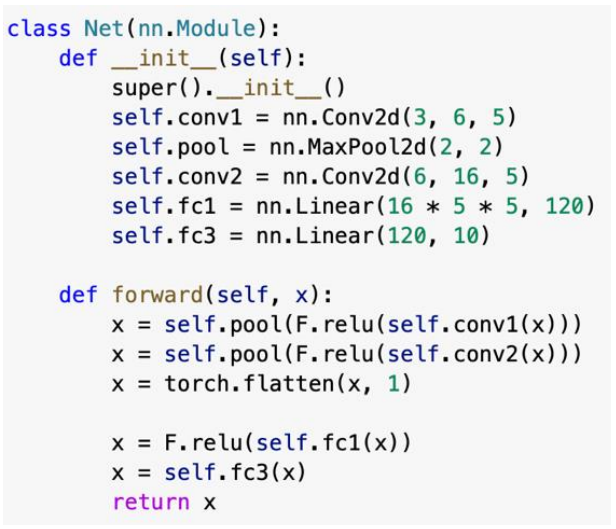
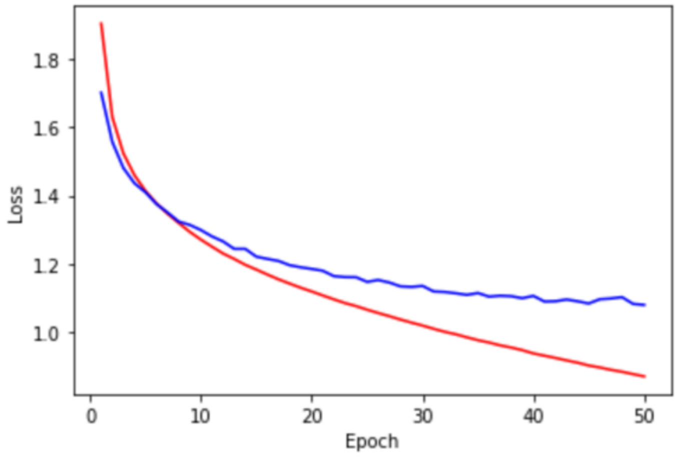
Architecture and Fine Tuning:

Baseline model:



```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc3 = nn.Linear(120, 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)

        x = F.relu(self.fc1(x))
        x = self.fc3(x)
        return x
```

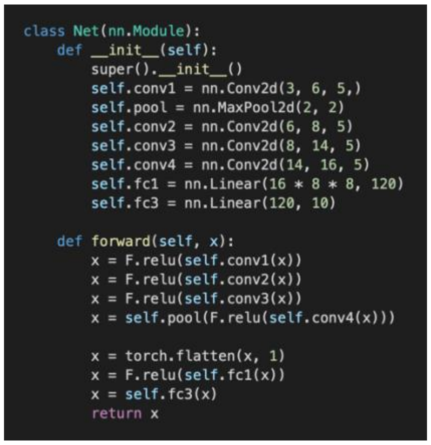


I originally started with the baseline CNN provided in the exercise. After a couple of tries I realized that the third FC layer was redundant. I therefore removed it. Further I changed the optimizer from SGD to the ADAM optimizer which seemed to converge faster and more consistently. Additionally, the initial batch size (4) lead to very slow convergence. I increased it to 32 allowing to take a full advantage of the GPU and allow an overall faster training. Lastly, I realized after many tries that a learning rate of 0.001 is too large and lead to inconsistent results. I thus changed it to 0.0001 which seemed to work.

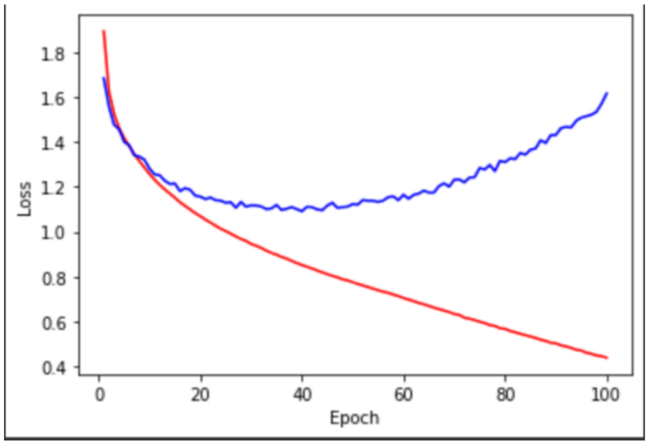
Note (graph above) that the training loss (Red) although very small doesn’t seem to converge. I therefore increased the number of epochs to 70. The disadvantage in doing so is the substantial increase in time waited.

Overfitting:
First, I realized that for small changes are enough to encourage relatively substantial changes in the network. I tried a couple of methods to induce overfitting in my model. Note none of the results exceeded the performance of my baseline model.

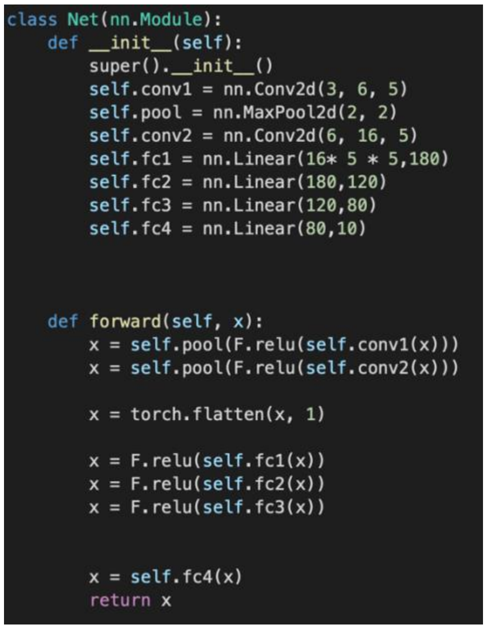
<u>Increase convolutional layers by an additional 2 layer as well as a corresponding ReLU function for each.</u> Note for these new layers we added a zero padding in order to keep the image dimensions the same as before. I also removed the one pooling layers (else unchanged). Note that removing pooling layers should promote overfitting.



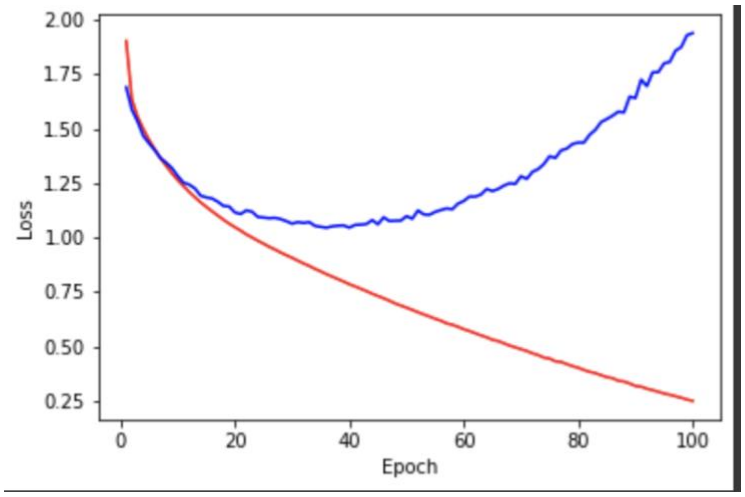
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5,)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 8, 5)
        self.conv3 = nn.Conv2d(8, 14, 5)
        self.conv4 = nn.Conv2d(14, 16, 5)
        self.fc1 = nn.Linear(16 * 8 * 8, 120)
        self.fc3 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.pool(F.relu(self.conv4(x)))

        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc3(x)
        return x
```



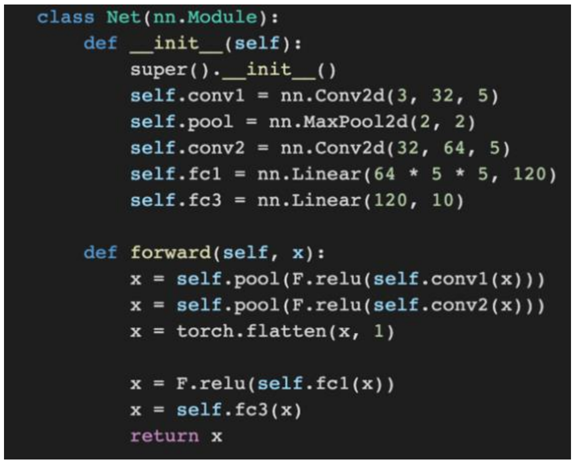
<u>Increase FC layers by an additional 3 layers (else unchanged).</u>



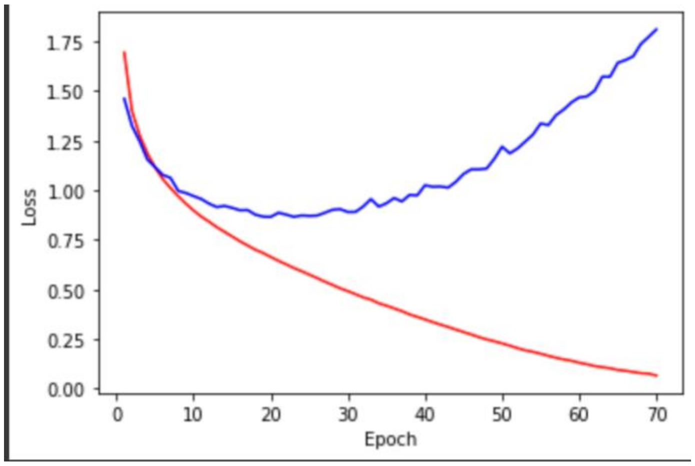
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16* 5 * 5,180)
        self.fc2 = nn.Linear(180,120)
        self.fc3 = nn.Linear(120,80)
        self.fc4 = nn.Linear(80,10)



    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)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))


        x = self.fc4(x)
        return x
```



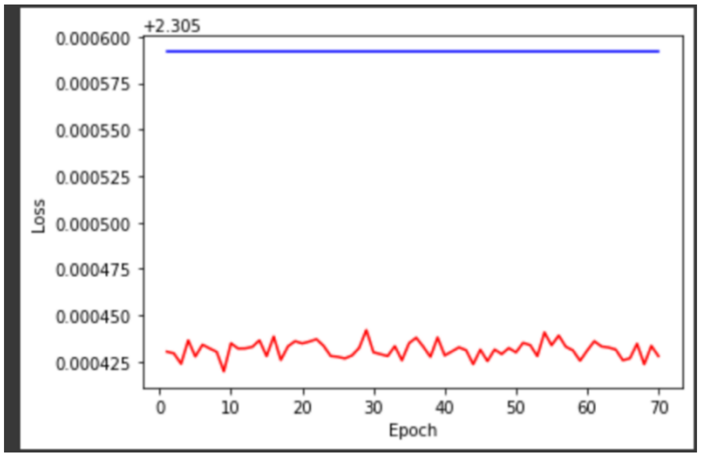
Increased the output channels of each convolutional layers to get 64 channels at the end. (Else unchanged). I got really good results. Let us note however, that after only 10 epochs the training and testing curves converged better and at a lower loss than the baseline model. Thus although there are more weights and it is harder to train, this model performs better and converges 10 times faster.



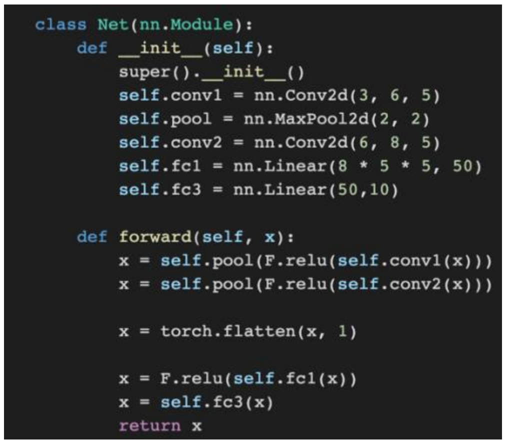
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64 * 5 * 5, 120)
        self.fc3 = nn.Linear(120, 10)

    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)

        x = F.relu(self.fc1(x))
        x = self.fc3(x)
        return x
```



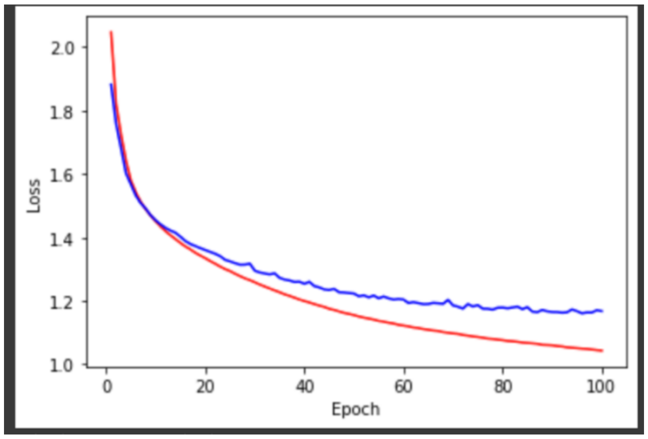
Increase activation maps to 64 and remove pooling layers. I must say that the results I got are inconsistent and I believe incorrect. I nevertheless decided to present them here. It seems like the model didn't train at all.



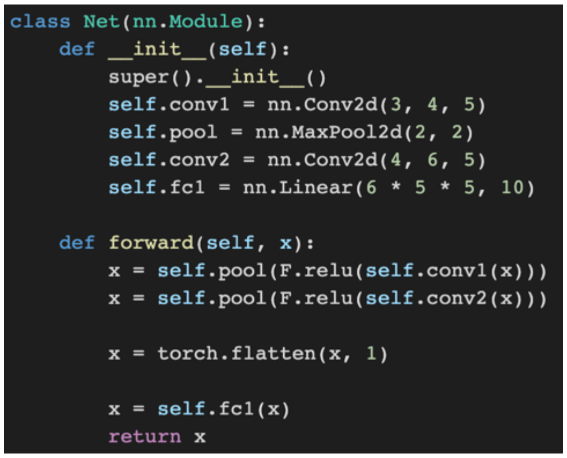
Underfitting:
I tried a couple of methods to get an underfitted model. The idea is to simplify the network.
I got very surprising results by removing one fully connected layer and reducing the number of total activation maps in the network. So it happens, I got identical results as the baseline model.
I seems our baseline model might be a little too big (connections)



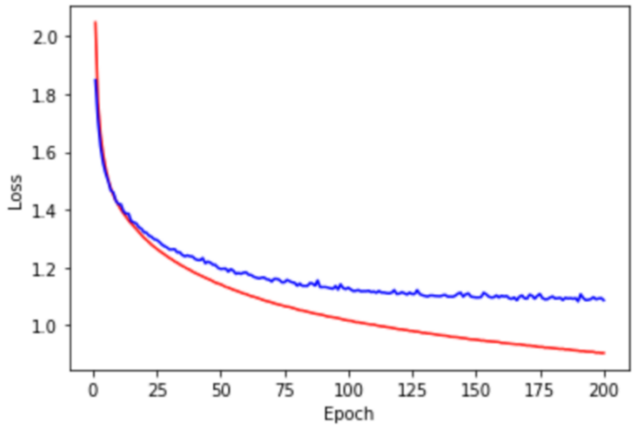
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 8, 5)
        self.fc1 = nn.Linear(8 * 5 * 5, 50)
        self.fc3 = nn.Linear(50,10)

    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)

        x = F.relu(self.fc1(x))
        x = self.fc3(x)
        return x
```



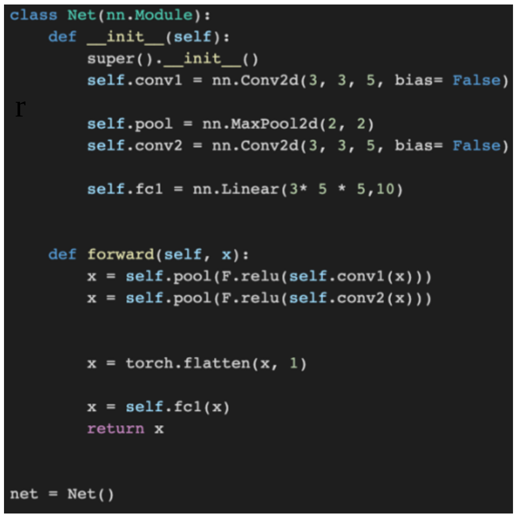
In order to induce underfitting I further simplified the model. I removed all Fully connected layer except the last one and further reduced the number of activation maps. **I received a comparable performance level to my base model again.** Note that since we used considerably less parameters, we can conclude that this model performs better than the baseline.
We will complete the rest of the exercise using this model



```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 4, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(4, 6, 5)
        self.fc1 = nn.Linear(6 * 5 * 5, 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)

        x = self.fc1(x)
        return x
```

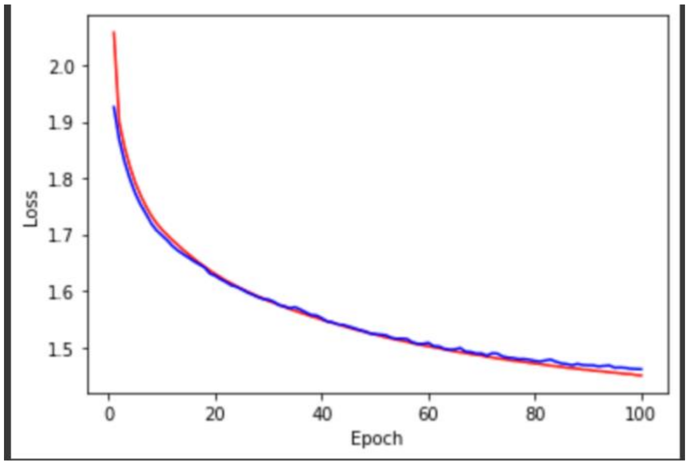


let’s take it further and see if we can get an underfitted model. In this case I kept the channel levels constant (three channels throughout the model) and removed the biases from the convolution function (bias = False). Here are the results: The model didn’t perform as well (higher loss value). I notice that as I simplify the network the correlation between the test curve and train curve increases. I can’t seem to figure why. (If you know please share!)

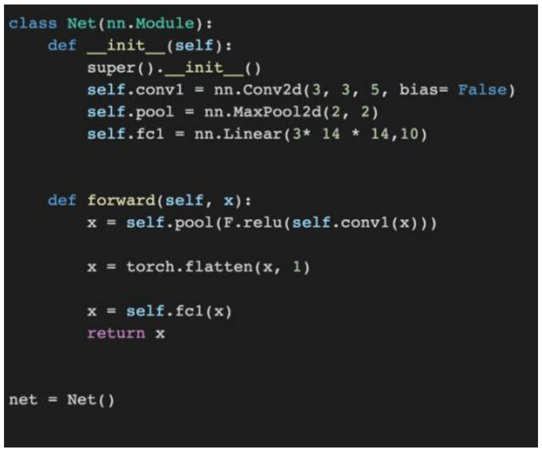


r

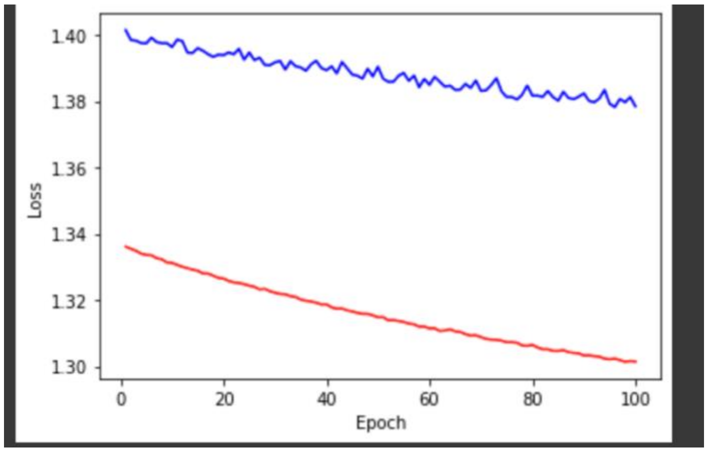
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 3, 5, bias= False)

        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 3, 5, bias= False)

        self.fc1 = nn.Linear(3* 5 * 5,10)


    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)

        x = self.fc1(x)
        return x


net = Net()
```



Finally, I simplified the network even further. As we can see the network didn’t learn at all.

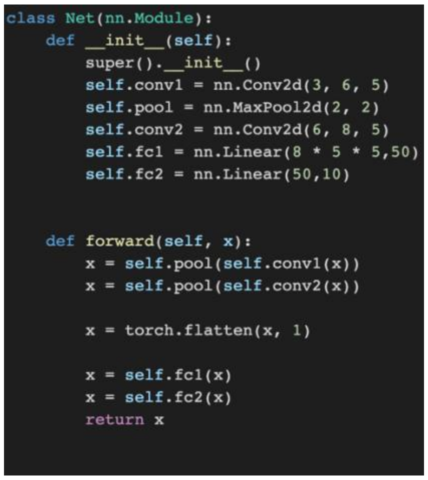


```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 3, 5, bias= False)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(3* 14 * 14,10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))

        x = torch.flatten(x, 1)

        x = self.fc1(x)
        return x


net = Net()
```

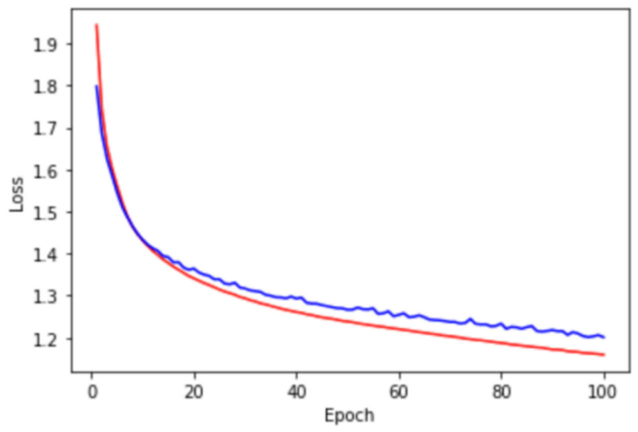


Linear vs. Non-Linearity.

I anticipate a reduction of the model’s performance since it becomes essentially the composition of linear functions -→matrix multiplication, although it is really an affine transformation if we consider the biases). We are supposed to have a reduction in the expressive power of the model as it becomes a simple linear operator. Therefore, as I noticed in my trials, increasing the network size although time consuming, doesn’t have any effect on performance.
Note we removed all the ReLU activation functions leaving the rest of the network untouched. Although max pooling is a non-linear operation, I chose to keep it to maintain the original structure of the Net.
After removing the non-linear components, the performance diminished.
However, I expected much worse performance and clear signs of underfitting. This may be an indication of the nature of the data. A more plausible theory is that maxpooling has a stronger non linerar effect that presumed.



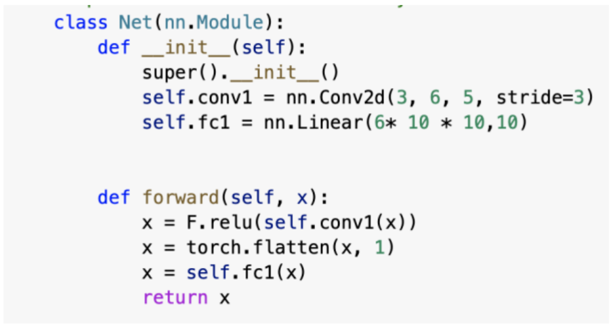
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 8, 5)
        self.fc1 = nn.Linear(8 * 5 * 5,50)
        self.fc2 = nn.Linear(50,10)


    def forward(self, x):
        x = self.pool(self.conv1(x))
        x = self.pool(self.conv2(x))

        x = torch.flatten(x, 1)

        x = self.fc1(x)
        x = self.fc2(x)
        return x
```



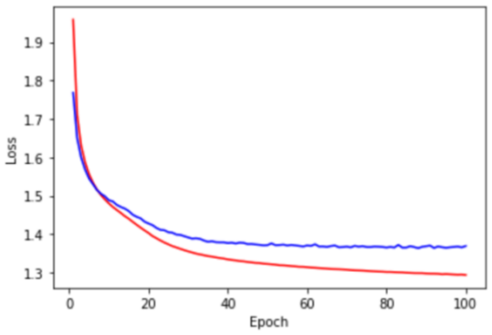
Cascaded Receptive Field

We will map the output of the first convolutional layer to the output layer using one FC layer. In order to avoid too large dimensions, we will play with the filter size as well as with the stride. Note that I decided to keep the number of activations maps the same since we already reduced it enough in the previous sections. Here are the results:
Note that the

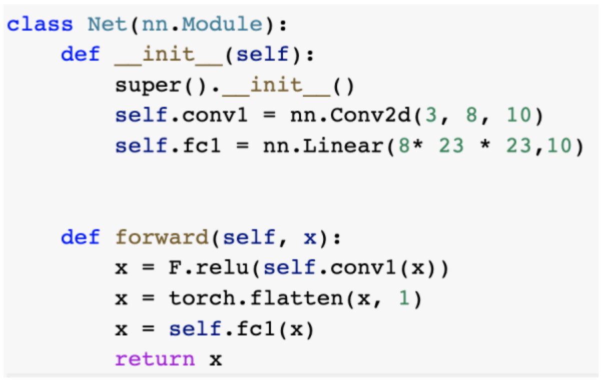
- One conv layer with stride 3 and one FC. (Kernel size unchanged)



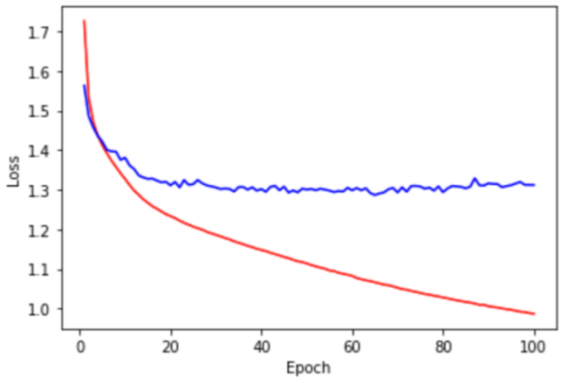
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, stride=3)
        self.fc1 = nn.Linear(6* 10 * 10,10)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        return x
```



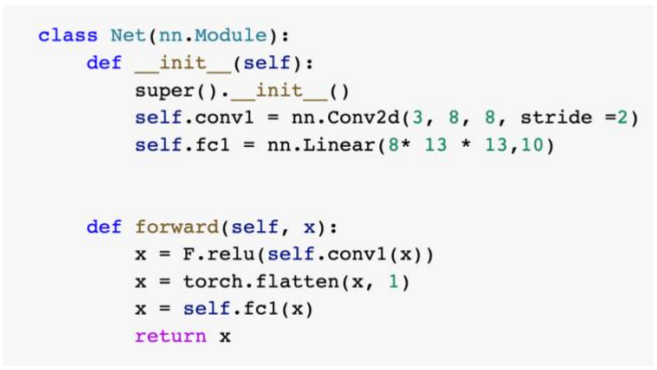
- One conv layer with kernel 10 and FC. (stride left unchanged)



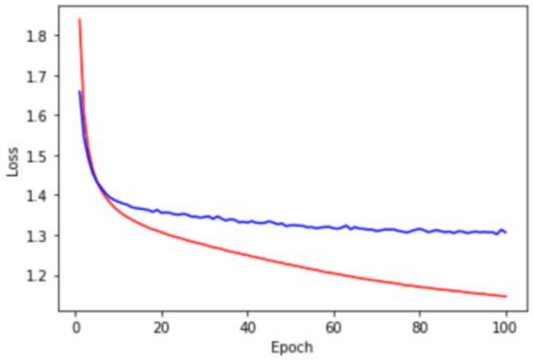
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 8, 10)
        self.fc1 = nn.Linear(8* 23 * 23,10)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        return x
```



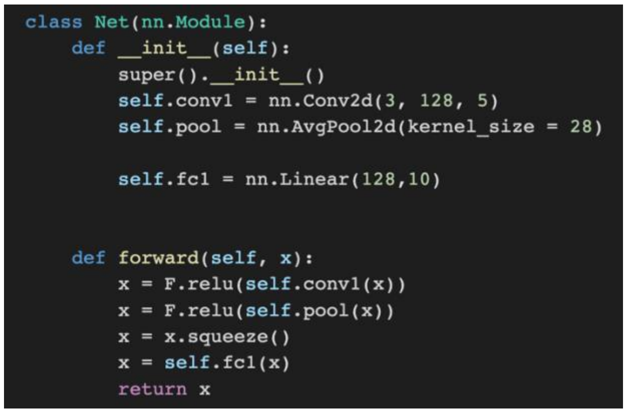
- One conv layer with stride 2 and kernel 8.



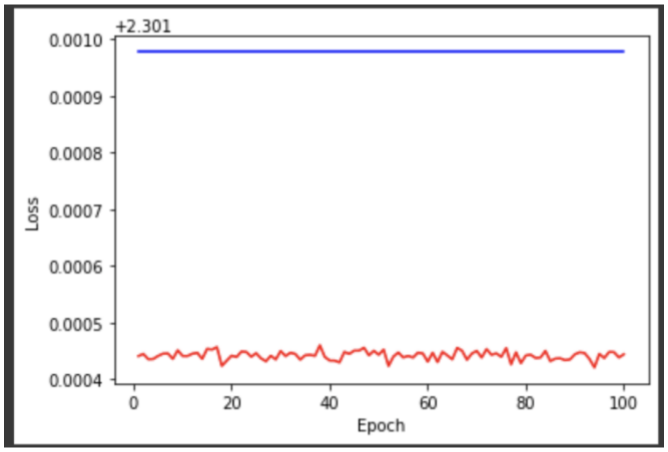
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 8, 8, stride =2)
        self.fc1 = nn.Linear(8* 13 * 13,10)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        return x
```



- We will implement a second method to make the model shallower but still maintain a large receptive field. We will do so by the use a “global average pooling” operator where you basically compute the average of the activations of each channel (i.e., the image x and y dimensions collapse) and then apply an FC layer. The idea is that FC layers are prone to overfitting and rely on regularizes like Dropout. In this way the correspondence between the features extracted and the output are not clear. We would want to use the spatial features extracted by the convolution layers to determined directly the outputs.
  Note In order to deal with the low dimension, we will compensate by increasing the channels to 128. Here are the results:
  Note the model performed catastrophically. This I think, is due to a lack of additional convolutional layers that are essential to form more complex filters.



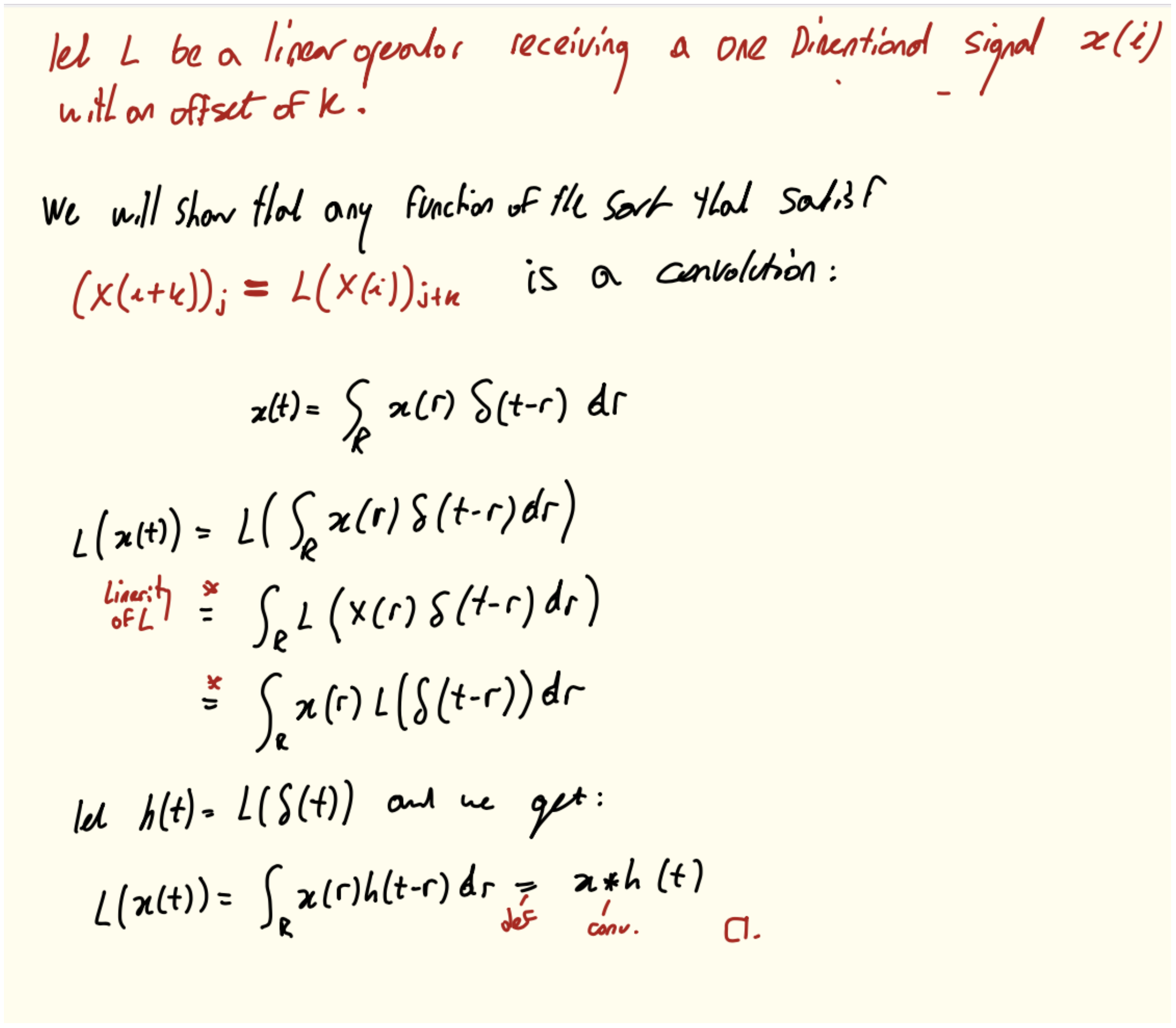
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 128, 5)
        self.pool = nn.AvgPool2d(kernel_size = 28)

        self.fc1 = nn.Linear(128,10)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.pool(x))
        x = x.squeeze()
        x = self.fc1(x)
        return x
```



.

Theoretical Questions:

Formally show that the condition L[x(i+k)](j) = L[x(i)](j+k) over a linear operator L receiving a 1D signal x(i) with offset k (the outer parentheses refer to the output signal), corresponds to a convolution. L operates over signals, i.e., it receives an input signal and outputs a signal, and the output signal index is denoted by j. Hint: decompose the signal x(i) into a weighted sum of translated delta functions, and then use the linearity of L. What input signal will output the underlining filter of the convolution?



let L be a linear operator receiving a one Dimentional signal $x(i)$ with an offset of k.

We will show that any function of the sort that satisfy

$(X(i+k))_j = L(X(i))_{j+k}$ is a convolution:

$$x(t) = \int_R x(r)\,\delta(t-r)\,dr$$

$$L(x(t)) = L\left(\int_R x(r)\,\delta(t-r)\,dr\right)$$

$$\underset{\text{of } L}{\text{Linearity}} \overset{*}{=} \int_R L\left(x(r)\,\delta(t-r)\,dr\right)$$

$$\overset{*}{=} \int_R x(r)\,L\left(\delta(t-r)\right)dr$$

let $h(t) = L(\delta(t))$ and we get:

$$L(x(t)) = \int_R x(r)\,h(t-r)\,dr \underset{\text{def}}{=} \underset{\text{conv.}}{x*h}\,(t) \quad \square.$$

When reshaping a 2D activation map (resulting from a convolutional layer) and feeding it into an FC layer, how important is the order (is there a good ordering or a bad ordering)? Why?

The order is completely unimportant. This is because a fully connected layer is **fully connected**. This means that the FC layer considers all connection from the previous layer and connects each of the neurons, regardless of the order, to all the neurons in the next layer. Regardless of the position of a particular neuron, the number of connections exiting it will be identical as well as the destinations.

The convolution operator is LTI (linear translation invariant). What about the following operators:

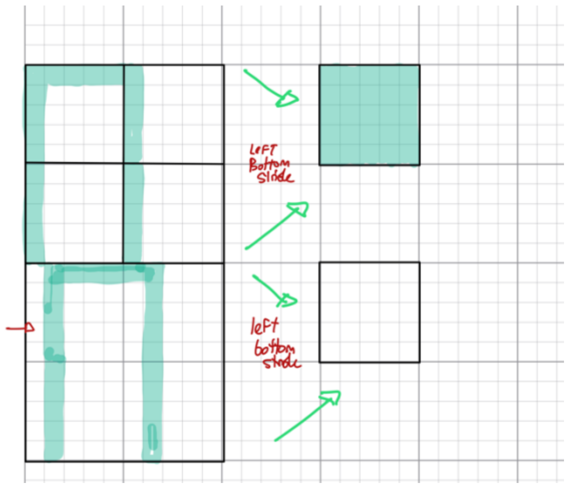
The ReLU activation function

The strided pooling layer (i.e., the one that always picks the top-right pixel in each block).

The addition of a bias

Explain your answers.

- ReLU: Not a linear operation and thus not LTI. PROOF: For x>0 , ReLU(-x)=0!=(-1)ReLU(x)
- Strided pooling layer: No it is not LTI. The stride pooling operation is obviously not space invariant. This can be illustrated by pooling over a specific form with a given stride and see the results:



- Addition of bias: This becomes an affine transformation! Thus, it is obviously not LTI. (A bias addition cannot be linear)