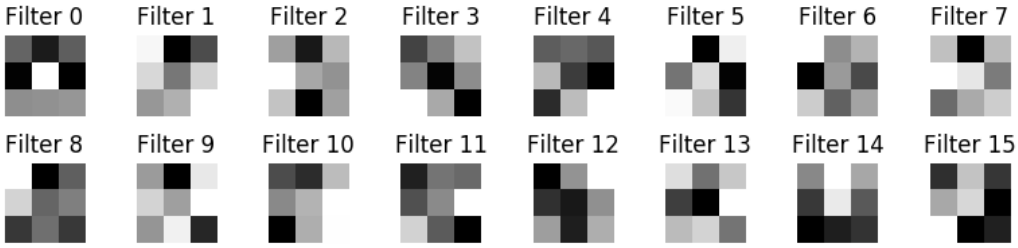
Homework 3

Ameer Hamza (ah18r)

March 30, 2020

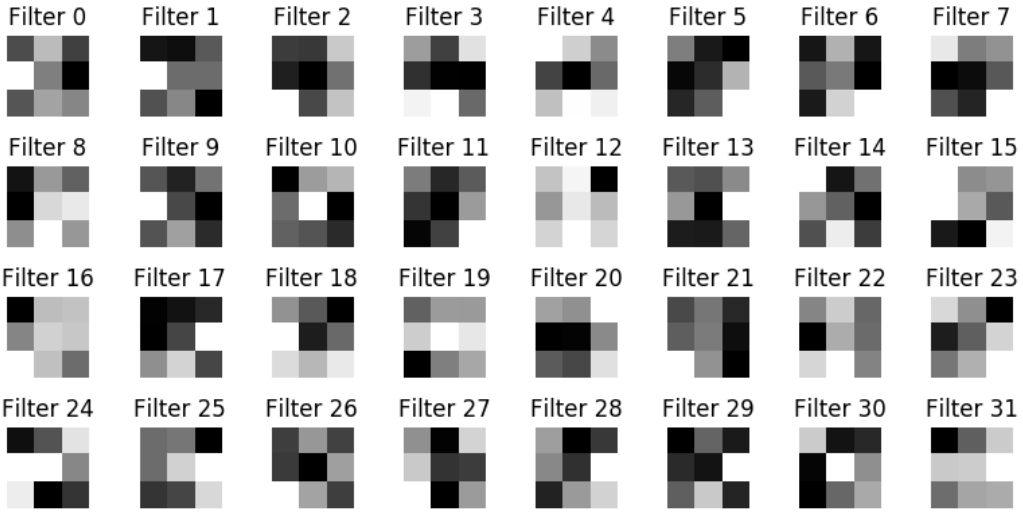
1 Question 1

1. My convolutional neural network contains 3 convolutional layers followed by a fully connected layer. First convolutional layer has 1 input channels and 16 output channels, and 3x3 filters. The following are the filters in the first layer. There is only one channel:

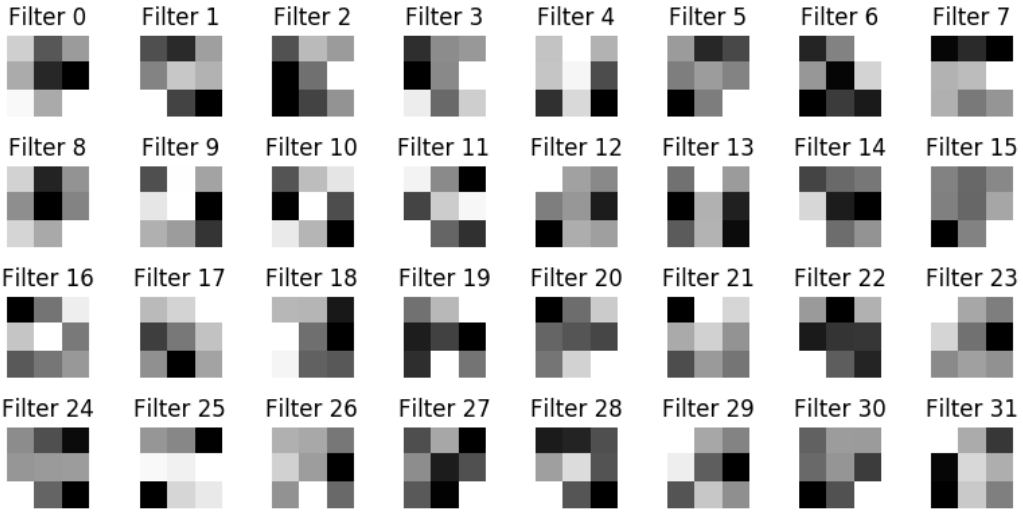


The second layer has 16 input channels, 32 output channels and 3x3 filters. Hence, it will have 32 filters of 3x3 size in each 16 channels. for simplicity, I will show filters for only first 2 channels, instead of 16.

Channel 0:



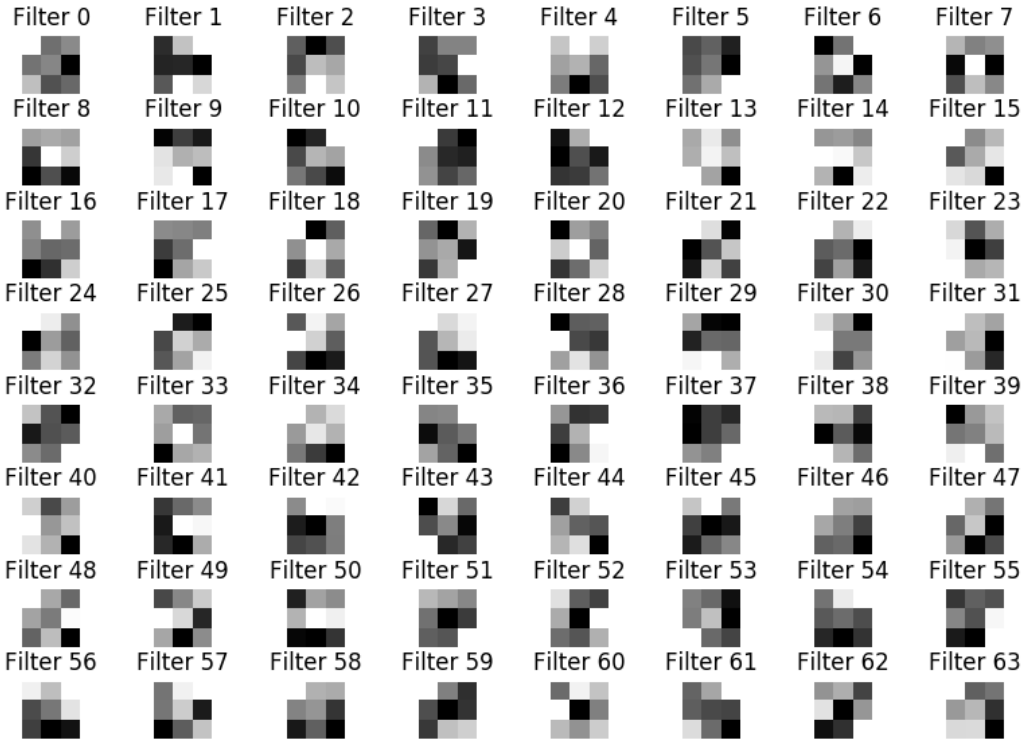
Channel 1:



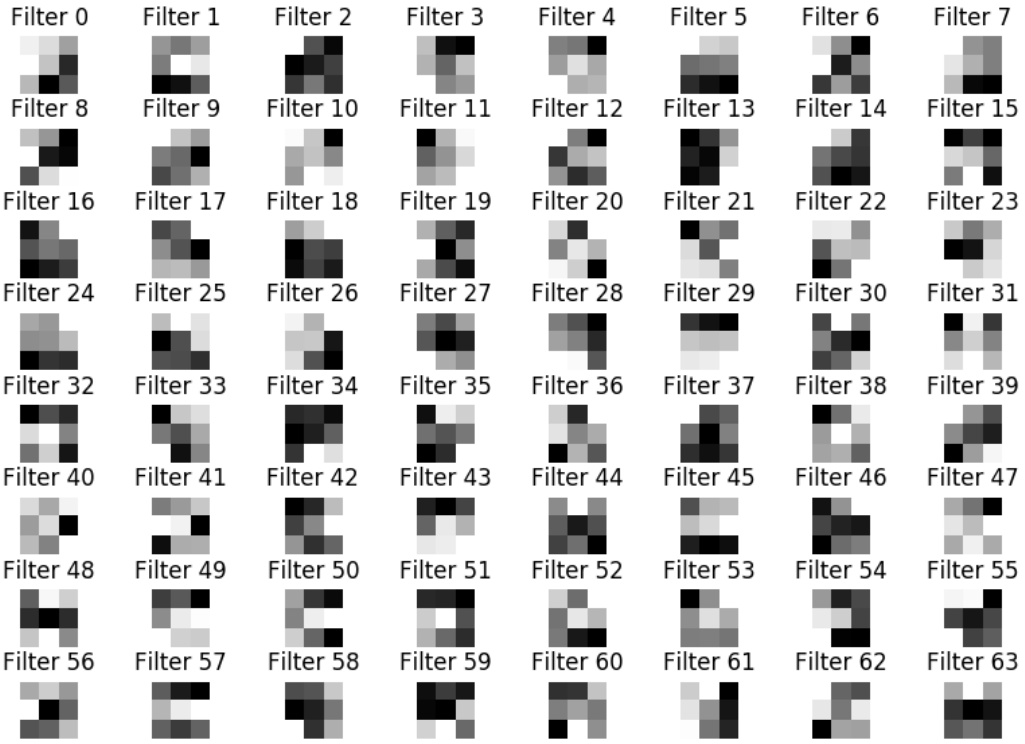
the third layer has 32 input channels, 64 output channels and 3x3 filters. hence, it will have 64 3x3 filters in each of 32 channels. Again, I will show

only show first 2.

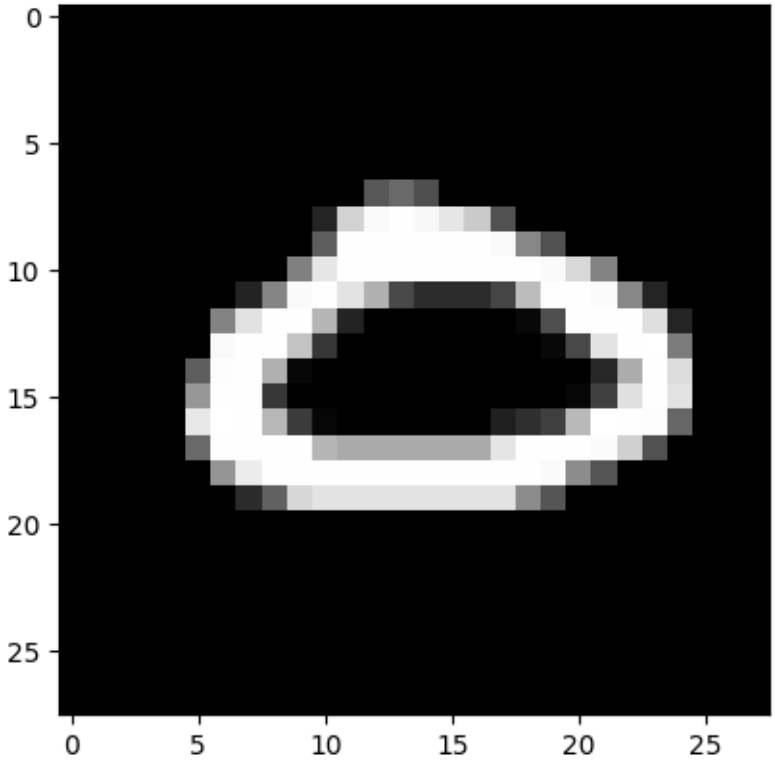
Channel 0:



Channel 1:

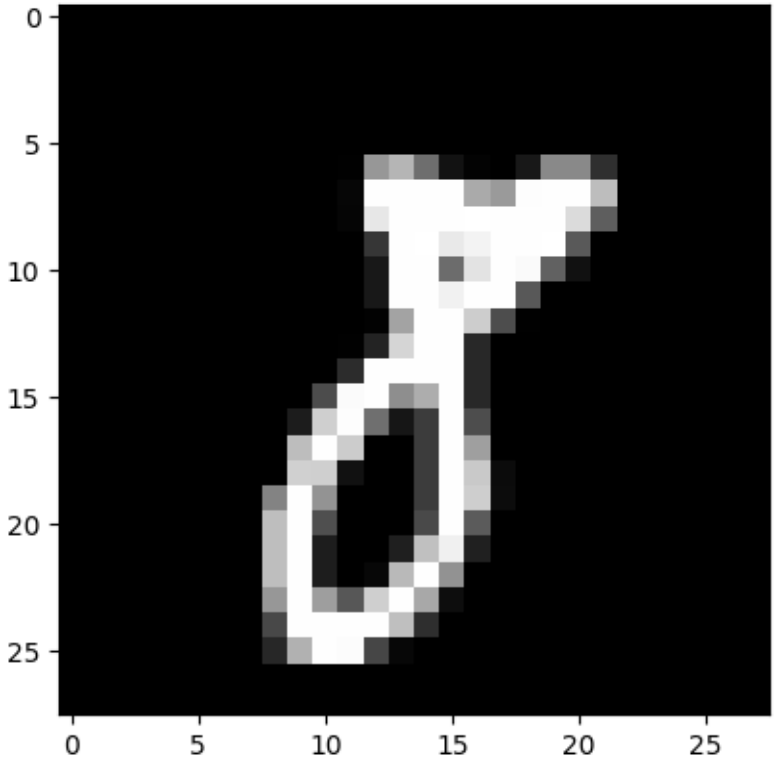


2. Here is the image of a data sample correctly classified as 0.

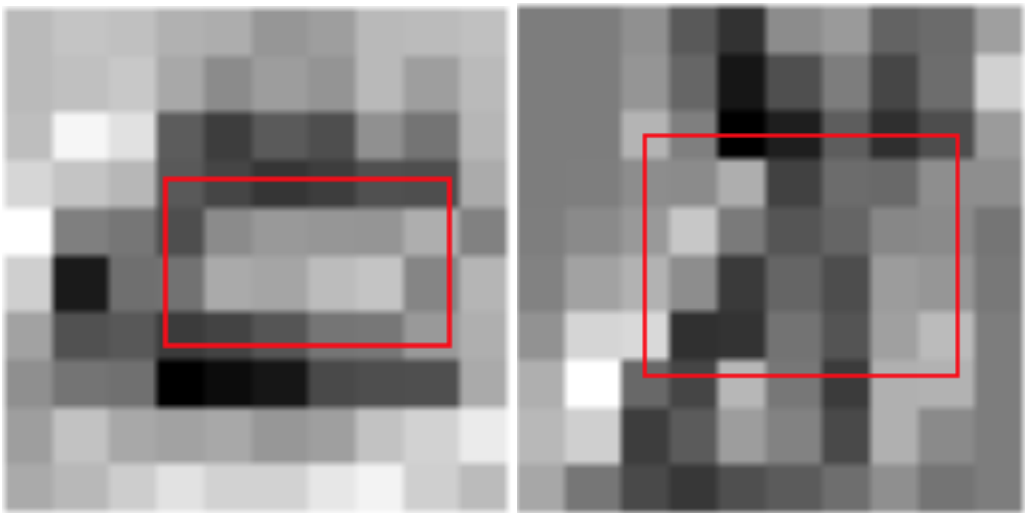


The feature maps for all the output channels look like this:

The image and feature maps for digit 8 are following:


0
5
10
15
20
25
0
5
10
15
20
25

By comparing the feature maps, it seems the center part of the feature maps (or digits) is important to distinguish. Zero has the center part empty while eight has a cross-like shape. Feature maps for the one same channel for both digits shown. First is 0, second is 8:



3. Even though most of the times the prediction should not change, sometimes it can change from 1. Here is a correct prediction:

Figure 1: Original

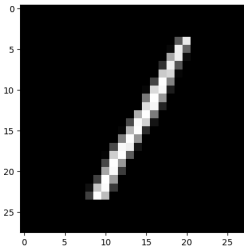


Figure 2: Left

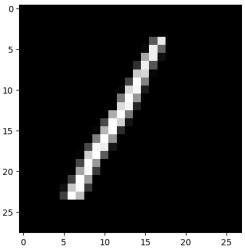
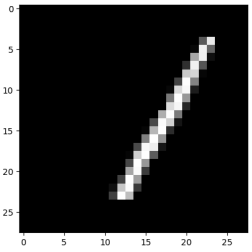


Figure 3: Right



However, here is a sample where the right shift predicts it as 7:

Figure 4: Original

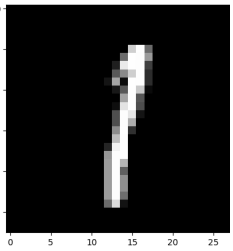


Figure 5: Left

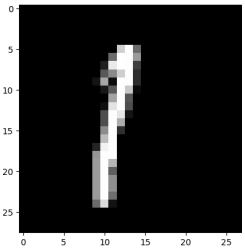
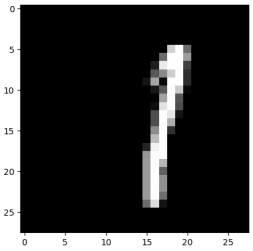


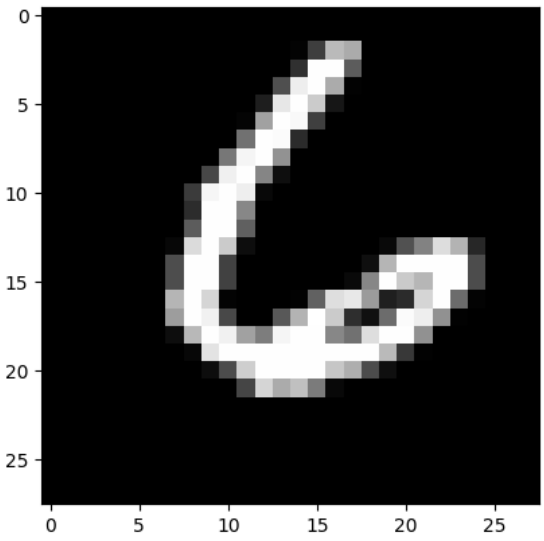
Figure 6: Right



Here, moving the data to right makes it think it is 7. The reason might be that the 7's lower half is usually at that position and since this one has a kind of edge on the top, that slightly resembles the upper half of 7, the model thinks it is actually 7. Hence, sometimes it can predict other letters but mainly it predicts 1.

2 Question 2

1. Here is the selected data sample that is 6:



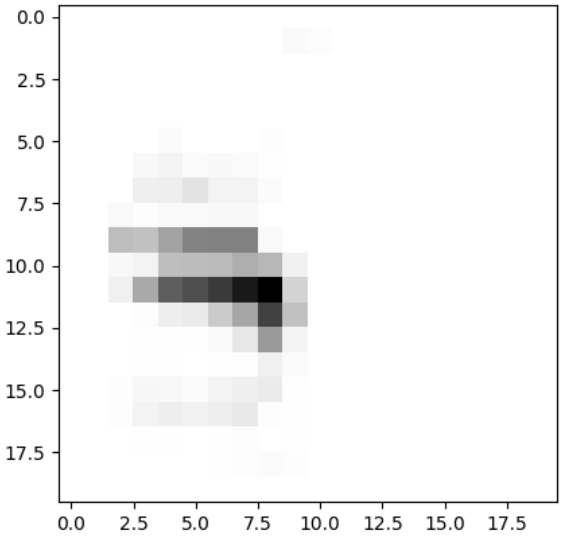


Figure 7: Probability of 6 map

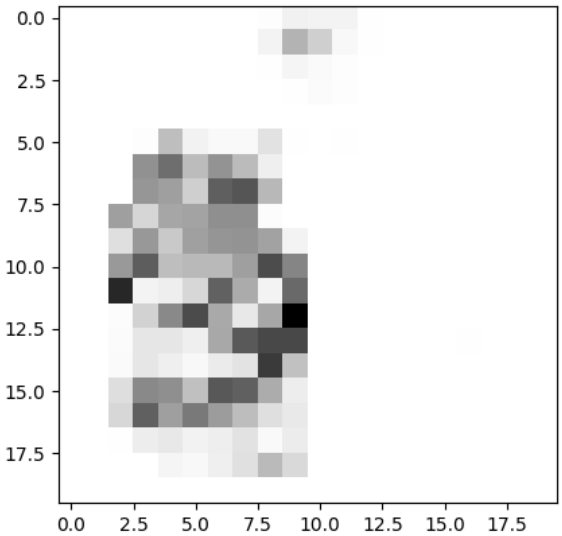


Figure 8: Highest Probability map

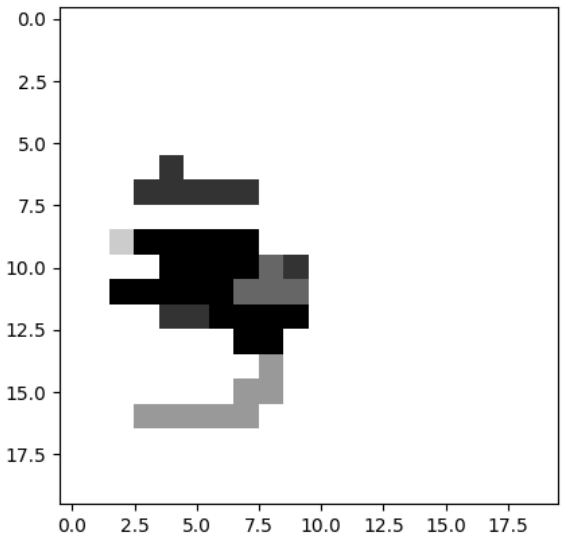
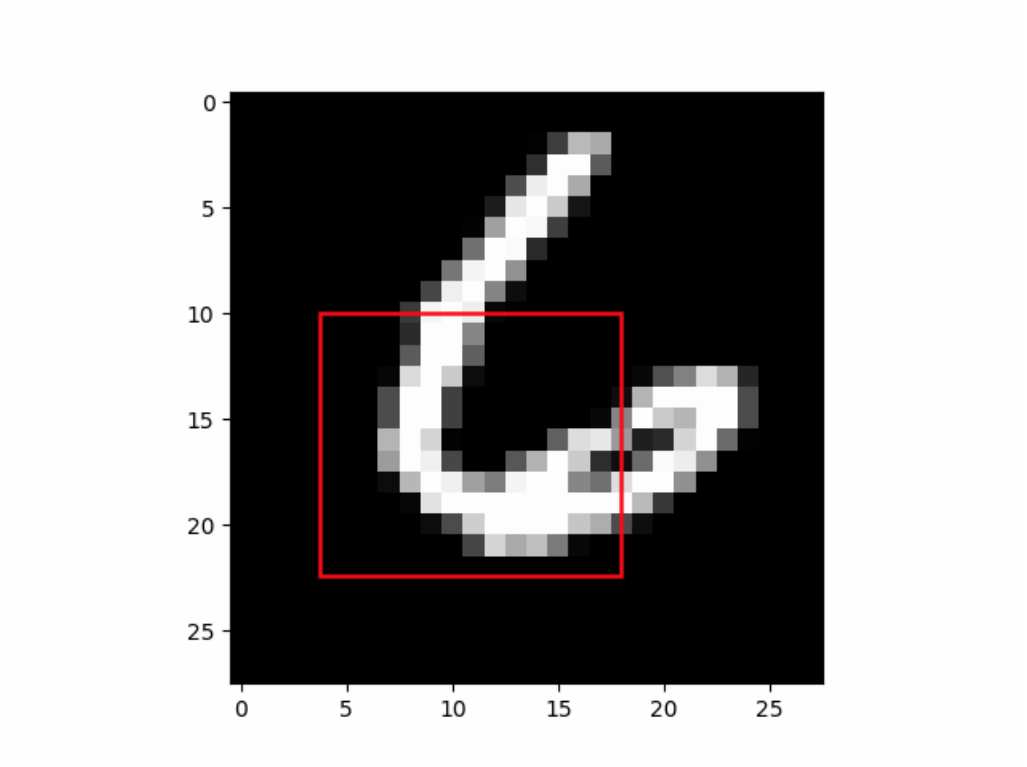


Figure 9: Label map

2. By analyzing the maps, it is clear that the curve part at the bottom, especially more to the left (as shown in the figure) is important for recognition.



3. Yes. We can make adversarial images by covering part of the image. Since the highlighted part of 6 image is important for predicting 6, that has to be replaced in order for it to not predict 6. To make it predict another digit like 5, take the part that is important for 5 to be predicted, and replace the important part of image 6 with that.

3 Question 3

1. Here are the outputs $\hat{y}^{(t)}$ for $t = 1 : 3$ and customized loss values.

```
y^(1) is tensor([[0.9492160082, 0.0507839993]])
y^(2) is tensor([[0.9522199035, 0.0477800518]])
y^(3) is tensor([[0.9400112629, 0.0599887557]])
loss is 3.0072081089019775
```

2. Approximate values of derivative of loss with respect to b1 and b2:

```
Approximate derivative of Loss wrt B1: -0.010728836059570312

Approximate derivative of Loss wrt B2: 0.432729721069335 94
```

3. Derivative of loss with respect to b1 comes out to be: -0.0099.
   Derivative of loss with respect to b2 comes out to be: 0.4340.

```
Derivative of a1^(1) wrt b1: 1
Derivative of a2^(1) wrt b1: 0
Derivative of h1^(1) wrt b1: 0.07065081596374512
Derivative of h2^(1) wrt b1: 0.0
Derivative of a1^(2) wrt b1: 1.0706508159637451
Derivative of a2^(2) wrt b1: 0.0
Derivative of h1^(2) wrt b1: 0.004500159993767738
Derivative of h2^(2) wrt b1: 0.0
Derivative of a1^(3) wrt b1: 1.004500150680542
Derivative of a2^(3) wrt b1: 0.0
Derivative of h1^(3) wrt b1: 0.010065767914056778
Derivative of h2^(3) wrt b1: 0.0
Derivative of loss wrt h1^(3): -0.9896358251571655
Derivative of loss wrt h2^(3): 0.9896358251571655
Derivative of Loss wrt b1: -0.00996144488453865

Derivative of a1^(1) wrt b2: 0
Derivative of a2^(1) wrt b2: 1
Derivative of h1^(1) wrt b2: 0.0
Derivative of h2^(1) wrt b2: 0.07065081596374512
Derivative of a1^(2) wrt b2: -0.07065081596374512
Derivative of a2^(2) wrt b2: 1.1413016319274902
Derivative of h1^(2) wrt b2: -0.0002969595370814204
Derivative of h2^(2) wrt b2: 0.012992793694138527
Derivative of a1^(3) wrt b2: -0.01328975334763527
Derivative of a2^(3) wrt b2: 1.025985598564148
Derivative of h1^(3) wrt b2: -0.00013317227421794087
Derivative of h2^(3) wrt b2: 0.4384220838546753
Derivative of loss wrt h1^(3): -0.9896358251571655
Derivative of loss wrt h2^(3): 0.9896358251571655
Derivative of Loss wrt b2: 0.43400999903678894
```

4. After running one step of gradient descent, the new values of b are:

```
tensor([-1.0000199080,  0.9991319776])
```

5. The loss value with new values of b, along with the difference from previous value:

```
loss is 3.006831407546997

Difference in the loss value: 0.00037670135498046875
```

4 Question 4

1. The reason here is the vanishing or exploding gradient problem. In case of vanishing gradient problem, when there are long sequence on input, it becomes difficult to learn and tune the parameters in the early layers in the network. This happens because of the non-linearity of activation function like tanh. These functions map a large range of values into a small region. The derivatives become so small that it becomes difficult to learn the early layer parameters. The product of derivatives can also explode when weights are large enough. Let's say for derivative wrt W:

$$\frac{\partial h^{(\tau)}}{\partial W} \propto W \tag{1}$$

   As we go through the sequence, the W keep getting accumulating. If W is large, the product of W's get really large and gradient explodes. Same goes for U and b. Hence, the parameters that are difficult to learn are W, U and b.

2. If we treat the network as echo state network, then we will fix the input weights (U) and the recurrent weights (W) (also the input bias b), while the output weights (V) (and the bias c) should be learned.

3. The RNN suffers from the vanishing/exploding gradient problem. LSTM solves this by introducing a solution that is called the forget term f. The equations are as follows:

$$C_t = f_t * C_{t-1} + g * \hat{C}_t \tag{2}$$

$$\hat{C}_t = \sigma(b + W.h_{t-1} + U.x_t) \tag{3}$$

Here, g is the input gate. As it is evident from the equations, only a fraction of the previous state is actually included in the current state. This helps with the vanishing/exploding gradients. If there is a problem with exploding gradients, the value of f is set close to 0 by the network. this way, the gradients do not explode due to chained multiplication. On the other hand, if the vanishing gradient problem is there, the value of f is set close to 1, hence helping with gradients becoming not too low.

5 Question 5

On an LSTM network with the same inputs and weights and with cells initialized as [0, 0], the values $\hat{y}_t$ for t=1:3 and customized loss come out to be:

```
y^(1) is tensor([[0.8694537282, 0.1305462867]])
y^(2) is tensor([[0.8758141398, 0.1241858527]])
y^(3) is tensor([[0.8315436244, 0.1684563607]])
loss is 1.8909997940063477
```

The derivative of loss with respect to B1 and B2 are given:

```
Derivative of Loss wrt B1: 0.0286102294921875
Derivative of Loss wrt B2: 0.4863739013671875
```

After running one step of gradient descent, the new loss value turns out to be:

```
new loss is 1.891475796699524
Difference in the loss value: -0.00047600269317626953
```

The derivative of loss wrt B1 and B2 is calculated using central difference method and not by unfolding the graph through time.