

Homework 4

600.482/682 Deep Learning

Spring 2019

March 17, 2019

Due Fri. 03/15 11:59pm.

Please submit a zip file including your runnable '.ipynb' jupyter notebook, name as 'First Name_Last Name_HW4.ipynb' to Gradescope "Homework4 - Zip Submission" with entry code MYRR74 and a separate PDF report to Homework 4 - Report.

You are supposed to program using PyTorch framework in Q2 and Q3.

1. In lecture 7, we have learned that a two-layer MLP can model polygonal decision boundaries. You are given data sampled from a "two-patagons" decision boundary (refer to lecture 7, slide 70). Points within this structure are "true" (output is 1) while points outside are considered "false" (output is 0). This decision boundary can be modeled with a total of $2 \times 5 + 2 + 1$ neurons with threshold activation function. There is no need to manually split the data, only consider results on this one single dataset for this toy example. The name of the data file is "HW4_data.npy". Please load this via np.load function. You will obtain a numpy array with the shape (60000,3). The first two columns are x, y coordinates and the third one represent the labels. The vertices of the polygons are (500, 1000), (300, 800), (400, 600), (600, 600), (700, 800), (500, 600), (100, 400), (300, 200), (700, 200), (900, 400) Plase use the first 50000 points as your training set and the rest as your testing set. Please

- (a) **Manually** setup an MLP with threshold activation* by selecting weights that can model the above decision boundary and verify that the all samples are classified correctly. Threshold activation is $y_i = 1 \cdot (w_i^T x + b_i > 0)$

Ans:

The Weights and Biases are selected as the following:

```
dic[0][0] = np.array([-1,1])
dic[0][1] = -5
dic[1][0] = np.array([1,2])
dic[1][1] = 14
dic[2][0] = np.array([1,0])
dic[2][1] = 6
dic[3][0] = np.array([1,-2])
dic[3][1] = -6
dic[4][0] = np.array([-1,-1])
dic[4][1] = -15
dic[5][0] = np.array([-1,0.5])
dic[5][1] = -3.5
dic[6][0] = np.array([1,1])
dic[6][1] = 5
dic[7][0] = np.array([1,0])
dic[7][1] = 2
dic[8][0] = np.array([1,-1])
dic[8][1] = -5
dic[9][0] = np.array([-1,-0.5])
dic[9][1] = -8.5
```

From the output number of correctly classified samples 60000, we verify that all samples are classified correctly.

- (b) Now, we have found one network model that can perfectly classify this problem. The above manual setup used threshold activations that are non-trainable. Consequently, please replace the 'threshold' with sigmoid activation,

$$y_i = \frac{1}{1 + e^{-(w_i^T x + b_i)}}$$

and change every node/perceptron such that its parameters can be trained. Randomize the parameters and use gradient descent to optimize the parameters. (Hint: random

initialization can be set using a uniform random between the largest and smallest parameter value in 1(a) you have set). In the last layer, consider an activation of > 0.5 as "true", i.e. output is 1. Please try multiple times (e.g. 5) and report your findings. Can you find a solution that performs as good as the above "true" solution?

Ans:

We tried 5 times, with each time running 100 iterations, initial step size as 0.1, step size decays to 1/10 every 20 iterations, and the following are the loss and accuracy of Training and Testing, respectively:

```
Train and Test with random initialization round 0:
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: overflow

Average Loss is: [[2.33107402]]
Train Accuracy: 82.282%
Average Loss is: [[0.65754397]]
Test Accuracy: 69.92%
Train and Test with random initialization round 1:
Average Loss is: [[2.33549455]]
Train Accuracy: 82.282%
Average Loss is: [[0.65702699]]
Test Accuracy: 69.92%
Train and Test with random initialization round 2:
Average Loss is: [[2.0577669]]
Train Accuracy: 82.282%
Average Loss is: [[0.59243162]]
Test Accuracy: 69.92%
Train and Test with random initialization round 3:
Average Loss is: [[2.33283494]]
Train Accuracy: 82.282%
Average Loss is: [[0.65645398]]
Test Accuracy: 69.92%
Train and Test with random initialization round 4:
Average Loss is: [[2.33261661]]
Train Accuracy: 82.282%
Average Loss is: [[0.65657701]]
Test Accuracy: 69.92%
```

We conclude that 5 tries all converges to the same train and test accuracy, however, the average loss of training and testing vary a little (changes mostly only occur in the 3rd decimal point). These Accuracy are much lower than 100% as the above "true" solution.

- (c) Now, define an MLP with increased capacity (try playing with a minimum of two different setups trading off increased depth with increased width) and see whether you can achieve higher classification accuracy.

Ans:

MODEL WITH INCREASED WIDTH: we increased the first layer width from 10 to 15, and the following are the accuracy and loss:

```
Train Accuracy: 82.282%
Average Loss is: [[0.65701566]]
Test Accuracy: 69.92%
```

The accuracy does not improve, due to the fact that the original width is already the optimal width for predicting this special dataset set by 10 different boundaries. The extra neurons in the first layer might not be very useful during the training process.

MODEL WITH INCREASED DEPTH: we increased the depth of our model by putting an extra layer containing 2 neurons before the last layer, and the followings are the accuracy and loss:

```
Train Accuracy: 82.282%
Average Loss is: [[0.6581213]]
Test Accuracy: 69.92%
```

The accuracy stays the same as all previous results, and the tiny increased in loss might be a variation in precision. However, the model with an increased depth supposedly should have a better accuracy. Reasons that this architecture does not work out could be the vanishing gradient of several sigmoid layers, the narrow range of the random initialization, as well as the limited number of neurons of our extra layer.

2. In this problem, we start to play with convolutional neural networks (CNN).

- (a) Please download Fashion MNIST official released dataset. (<https://github.com/zalandoresearch/fashion-mnist>). Design a small CNN (e.g. 3-4 hidden layers) using the tools we encountered in class (e.g. convolution and pooling layers, activation functions, regularization). **Please think hard for yourself, do not talk with your colleagues nor use code available online. Please design ahead with what you think makes sense. This is NOT about performance.**

Ans:

The designed small CNN has 3 Conv layers and 1 fully connected layer. Max-Pool2d is used as pooling layers, and ReLU is used as activation functions. The design is as the following:

```
##### (a)-continued #####
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.ly1 = nn.Sequential(
            nn.Conv2d(1, 4, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.ly2 = nn.Sequential(
            nn.Conv2d(4, 8, 4, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.ly3 = nn.Sequential(
            nn.Conv2d(8, 16, 5, 1, 3),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.linear = nn.Linear(256, 10)

    def forward(self, x):
        x = self.ly1(x)
        x = self.ly2(x)
        x = self.ly3(x)
        x = x.view(x.size(0), -1)
        x = self.linear(x)
        return x
```

- (b) Train your network and report the results on the test data of Fashion MNIST.

Ans:

The following are the 10 epoches' training loss and the final test accuracy is **87.69%**.

```
Epoch: 1
Training Loss: 0.3891
Epoch: 2
Training Loss: 0.3929
Epoch: 3
Training Loss: 0.4511
Epoch: 4
Training Loss: 0.3956
Epoch: 5
Training Loss: 0.5119
Epoch: 6
Training Loss: 0.2411
Epoch: 7
Training Loss: 0.3229
Epoch: 8
Training Loss: 0.2086
Epoch: 9
Training Loss: 0.3700
Epoch: 10
Training Loss: 0.3525
Test Accuracy is: 87.69%
```

- (c) Compare your architecture and results to at least one (preferred two) colleagues in our class. **Briefly** state the differences between your and your colleagues' architectures and results, and reason about why one or the other may perform better.

Ans:

The architecture and the results are as the following:

```
[ ] 1 # 3x3 convolution
2 def conv3x3(in_channels, out_channels, stride=1):
3     return nn.Conv2d(in_channels, out_channels, kernel_size=3,
4                       stride=stride, padding=1, bias=True)

[ ] 1 class ResidualBlock(nn.Module):
2     def __init__(self, in_channels, out_channels, stride=1, downsample=None):
3         super(ResidualBlock, self).__init__()
4         self.conv1 = conv3x3(in_channels, out_channels, stride)
5         self.pool1 = nn.MaxPool2d(2, 2)
6         self.bn1 = nn.BatchNorm2d(out_channels)
7         self.relu = nn.ReLU(inplace=True)
8         self.conv2 = conv3x3(out_channels, out_channels)
9         self.pool2 = nn.MaxPool2d(2, 2)
10        self.bn2 = nn.BatchNorm2d(out_channels)
11        self.downsample = downsample
12
13    def forward(self, x):
14        residual = x
15        out = self.conv1(x)
16        out = self.pool1(out)
17        out = self.bn1(out)
18        out = self.relu(out)
19        out = self.conv2(out)
20        out = self.pool2(out)
21        out = self.bn2(out)
22        if self.downsample:
23            residual = self.downsample(x)
24        out += residual
25        out = self.relu(out)
```

validation accuracy: 83.1000%

[18] Epoch 6
Loss: 0.4392 60000 / 60000

Training loss: 0.4392
Validation loss: 0.4521
Validation accuracy: 82.8900%

Epoch 7
Loss: 0.4395 60000 / 60000

Training loss: 0.4395
Validation loss: 0.4331
Validation accuracy: 83.4200%

Epoch 8
Loss: 0.4269 60000 / 60000

Training loss: 0.4269
Validation loss: 0.4296
Validation accuracy: 84.1000%

Epoch 9
Loss: 0.4104 60000 / 60000

Training loss: 0.4104
Validation loss: 0.4211
Validation accuracy: 84.4600%

This architecture performs slightly worse, potentially due to the fact that it only have two convolution layers with smaller kernel size. Therefore, its architecture simply has much smaller capacity to learn from training data.

- (d) Now, design an improved architecture based on your discussions and try to improve on your (and your colleagues') performance: Implement at least one of dropout and batch norm, use augmentation, play with optimizers, and try to beat all previous scores. Report your best results on the test set of Fashion MNIST. Try to compare again to your colleagues' results.

Ans: I simply add BatchNorm2d to each Convolution layer, and a dropout at the last convolution Layer:

```

Epoch: 1
Training Loss: 0.5951
Epoch: 2
Training Loss: 0.3648
Epoch: 3
Training Loss: 0.2870
Epoch: 4
Training Loss: 0.2950
Epoch: 5
Training Loss: 0.3155
Epoch: 6
Training Loss: 0.4100
Epoch: 7
Training Loss: 0.1763
Epoch: 8
Training Loss: 0.2107
Epoch: 9
Training Loss: 0.2153
Epoch: 10
Training Loss: 0.2289
Test Accuracy is: 89.69%

```

```

##### (c) #####
class CNN2(nn.Module):
    def __init__(self):
        super(CNN2, self).__init__()
        self.ly1 = nn.Sequential(
            nn.Conv2d(1, 4, 3, 1, 1),
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.ly2 = nn.Sequential(
            nn.Conv2d(4, 8, 4, 1, 2),
            nn.BatchNorm2d(8),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.ly3 = nn.Sequential(
            nn.Conv2d(8, 16, 5, 1, 3),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.MaxPool2d(2)
        )
        self.linear = nn.Linear(256, 10)

    def forward(self, x):
        x = self.ly1(x)
        x = self.ly2(x)
        x = self.ly3(x)
        x = x.view(x.size(0), -1)
        x = self.linear(x)
        return x

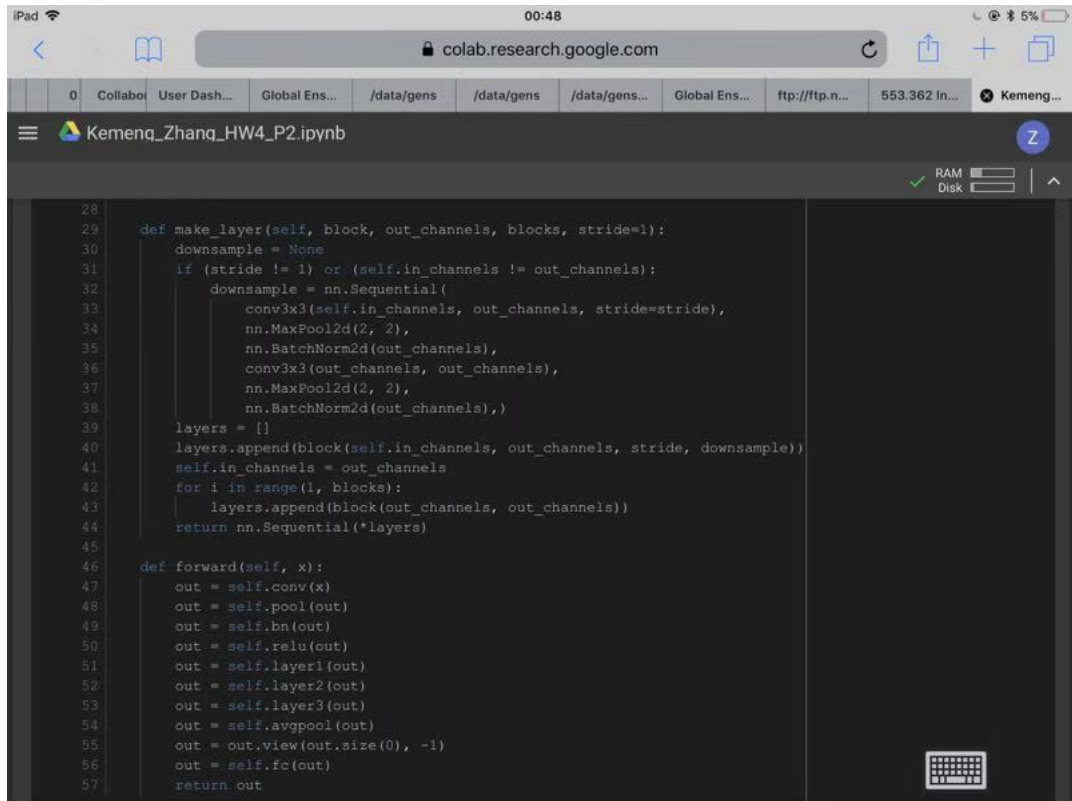
```

My friend also include extra layer of batchnorm and avgpool at the end.

```

[ ] 1 # ResNet
2 class ResNet(nn.Module):
3     def __init__(self, block, layers, num_classes=10):
4         super(ResNet, self).__init__()
5         self.in_channels = 3
6         self.conv = conv3x3(1, 3)
7         self.pool = nn.MaxPool2d(2, 2)
8         self.bn = nn.BatchNorm2d(3)
9         self.relu = nn.ReLU(inplace=True)
10        self.layer1 = self.make_layer(block, 6, layers[0])
11        self.layer2 = self.make_layer(block, 16, layers[1], 2)
12        self.layer3 = self.make_layer(block, 32, layers[2], 2)
13        self.avgpool = nn.AdaptiveAvgPool2d(1)
14        self.fc = nn.Linear(32, num_classes)
15
16        for m in self.modules():
17            if isinstance(m, nn.Conv2d):
18                nn.init.kaiming_normal_(m.weight, mode='fan_out',
19                                       nonlinearity='relu')
20                if m.bias is not None:
21                    nn.init.normal_(m.bias.data)
22            elif isinstance(m, nn.Linear):
23                nn.init.kaiming_normal_(m.weight.data)
24                nn.init.normal_(m.bias.data)
25            elif isinstance(m, nn.BatchNorm2d):
26                nn.init.constant_(m.weight, 1)
27                nn.init.constant_(m.bias, 0)
28
29        def make_layer(self, block, out_channels, blocks, stride=1):
30            downsample = None

```



```
28
29 def make_layer(self, block, out_channels, blocks, stride=1):
30     downsample = None
31     if (stride != 1) or (self.in_channels != out_channels):
32         downsample = nn.Sequential(
33             conv3x3(self.in_channels, out_channels, stride=stride),
34             nn.MaxPool2d(2, 2),
35             nn.BatchNorm2d(out_channels),
36             conv3x3(out_channels, out_channels),
37             nn.MaxPool2d(2, 2),
38             nn.BatchNorm2d(out_channels),)
39     layers = []
40     layers.append(block(self.in_channels, out_channels, stride, downsample))
41     self.in_channels = out_channels
42     for i in range(1, blocks):
43         layers.append(block(out_channels, out_channels))
44     return nn.Sequential(*layers)
45
46 def forward(self, x):
47     out = self.conv(x)
48     out = self.pool(out)
49     out = self.bn(out)
50     out = self.relu(out)
51     out = self.layer1(out)
52     out = self.layer2(out)
53     out = self.layer3(out)
54     out = self.avgpool(out)
55     out = out.view(out.size(0), -1)
56     out = self.fc(out)
57     return out
```

My Accuracy increased by 2% to 89.69%, and my friend's accuracy increased more than 6%, due to the extra convolution layer he added.

3. Supervised v.s. unsupervised learning. Please again use Fashion MNIST dataset.

- (a) An auto-encoder like structure is one of the most popular CNN architectures in computer vision. The encoding part reduces the original input dimension into a low-dimensional "code" (feature) while the decoding part expands this representation back into its original form (the image) as best as possible. The reason why this structure has gained popularity is because good filter kernels can be learned even though no labels (annotations) are available, because the target task is reconstruction of the input image albeit passing through a data-bottleneck. Learning without annotations is known as unsupervised learning. We have provided the implementation of such autoencoder together with this homework. Please refer to the instructions in Appendix. Train the provided autoencoder on the Fashion MNIST dataset and visualize the feature maps extracted in the first layer (closest to the input).

Ans:

Appendix 3.2 Question: For the training and testing dataset, I wrote my own data loader

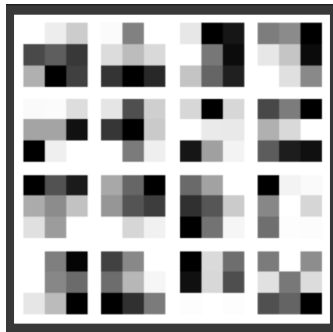
The training process is the following:

```

Epoch: 1
Training Loss: 0.1142
Epoch: 2
Training Loss: 0.1048
Epoch: 3
Training Loss: 0.0835
Epoch: 4
Training Loss: 0.0928
Epoch: 5
Training Loss: 0.0888
Epoch: 6
Training Loss: 0.0796
Epoch: 7
Training Loss: 0.0843
Epoch: 8
Training Loss: 0.0810
Epoch: 9
Training Loss: 0.0754
Epoch: 10
Training Loss: 0.0707
Epoch: 11
Training Loss: 0.0717
Epoch: 12
Training Loss: 0.0760
Epoch: 13
Training Loss: 0.0770
Epoch: 14
Training Loss: 0.0711
Epoch: 15
Training Loss: 0.0804
Epoch: 16
Training Loss: 0.0679
Epoch: 17
Training Loss: 0.0649
Epoch: 18
Training Loss: 0.0677
Epoch: 19
Training Loss: 0.0741
Epoch: 20
Training Loss: 0.0710

```

The visualize the feature maps extracted in the first layer:



- (b) These feature maps are likely noisy, and we hope to improve on these by training a denoising auto-encoder. The network structure of the denoising autoencoder remains the same as above, however, you will need to work with data augmentation of your inputs: In particular, please corrupt your input images by adding strong noise (Gaussian, Salt and Pepper, completely set pixels to zero) **but** keep the output (the target you are trying to reconstruct) unchanged. Please train your network again and visualize the feature maps. Do you see any difference?

Ans:

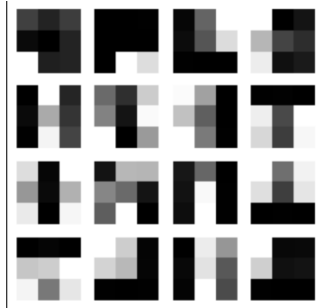
The training process is the following:

```

Epoch: 1
Training Loss: 0.3223
Epoch: 2
Training Loss: 0.2495
Epoch: 3
Training Loss: 0.2465
Epoch: 4
Training Loss: 0.2213
Epoch: 5
Training Loss: 0.2268
Epoch: 6
Training Loss: 0.2242
Epoch: 7
Training Loss: 0.2206
Epoch: 8
Training Loss: 0.2047
Epoch: 9
Training Loss: 0.1943
Epoch: 10
Training Loss: 0.1996
Epoch: 11
Training Loss: 0.2321
Epoch: 12
Training Loss: 0.1949
Epoch: 13
Training Loss: 0.2114
Epoch: 14
Training Loss: 0.2006
Epoch: 15
Training Loss: 0.1923
Epoch: 16
Training Loss: 0.1902
Epoch: 17
Training Loss: 0.1787
Epoch: 18
Training Loss: 0.2017
Epoch: 19
Training Loss: 0.1847
Epoch: 20
Training Loss: 0.2050

```

The visualize the feature maps extracted in the first layer:



There have clearly been many lower value (darker) pixels in each feature maps, potentially due to the 3 layers of heavy noises (Gaussian, Salt and Pepper, completely set pixels to zero) that affect on it.

- (c) With the above trained model, please freeze your autoencoder layers and "cut off" the decoding portion of the autoencoder. Connect the last layer of the encoding branch to a fully connected layer, and train the parameters of this last fully connected layer to be able to perform classification. Compare the performance of this model to your result in Problem 2.

Ans:

The model performs slightly worse than my previous model in Problem 2 (which yields an accuracy of 89.69%), potentially due to the smaller kernel size, and the strong noise added to the image. During the training, the loss remains very high after 20 epoches. The loss never truly converge, at the end of 100 epoches, we get a loss of 0.3656 with an accuracy of 84.86%, which is about the average.

| | |
|-----------------------|-----------------------|
| Epoch: 1 | |
| Training Loss: 0.9659 | |
| Epoch: 2 | |
| Training Loss: 0.8256 | |
| Epoch: 3 | |
| Training Loss: 0.6878 | |
| Epoch: 4 | |
| Training Loss: 0.7947 | |
| Epoch: 5 | |
| Training Loss: 0.4718 | |
| Epoch: 6 | |
| Training Loss: 0.6566 | |
| Epoch: 7 | |
| Training Loss: 0.6204 | |
| Epoch: 8 | |
| Training Loss: 0.6465 | |
| Epoch: 9 | |
| Training Loss: 0.4621 | |
| Epoch: 10 | |
| Training Loss: 0.4452 | Epoch: 90 |
| Epoch: 11 | Training Loss: 0.3942 |
| Training Loss: 0.4721 | Epoch: 91 |
| Epoch: 12 | Training Loss: 0.4625 |
| Training Loss: 0.5438 | Epoch: 92 |
| Epoch: 13 | Training Loss: 0.3913 |
| Training Loss: 0.6882 | Epoch: 93 |
| Epoch: 14 | Training Loss: 0.3078 |
| Training Loss: 0.4696 | Epoch: 94 |
| Epoch: 15 | Training Loss: 0.3271 |
| Training Loss: 0.3935 | Epoch: 95 |
| Epoch: 16 | Training Loss: 0.4344 |
| Training Loss: 0.5451 | Epoch: 96 |
| Epoch: 17 | Training Loss: 0.3507 |
| Training Loss: 0.6537 | Epoch: 97 |
| Epoch: 18 | Training Loss: 0.3451 |
| Training Loss: 0.5064 | Epoch: 98 |
| Epoch: 19 | Training Loss: 0.2331 |
| Training Loss: 0.3578 | Epoch: 99 |
| Epoch: 20 | Training Loss: 0.4032 |
| Training Loss: 0.4433 | Epoch: 100 |
| | Training Loss: 0.3656 |

| | |
|--------------------------|--------------------------|
| Test Accuracy is: 82.09% | Test Accuracy is: 84.86% |
|--------------------------|--------------------------|

(d) Improvement of structure

Ans:

Improved architecture can be found in `Xi_Wang_HW4_Q3_Improved.ipynb`. Several attempts have been made to improve the original sample architecture, including changing the batch size from 128 to 64, and changing the encoder's convolution architecture. Specifically, one extra layer is added before the first layer, with stride of 1, kernel size and output volume stays the same. This extra layer is followed by a max pool layer of stride 1. After that, the previous first layer's stride is changed from 3 to 2 to match the output, and the last layer takes an input volume of 8 and output a volume of 8. The effort of this architecture is trying to preserve the original image feature by extracting convolution layers of exact same size (28x28). A slight decrease in loss is observed in training from part (a) and (b). Loss in (b) and (c) is more obvious, and the loss is increased by 2.5% by Epoch 10, and 4% by Epoch 100, which we will show here:

| | |
|--------------------------|--------------------------|
| Epoch: 1 | |
| Training Loss: 0.2001 | |
| Epoch: 2 | |
| Training Loss: 0.2167 | |
| Epoch: 3 | |
| Training Loss: 0.2117 | |
| Epoch: 4 | |
| Training Loss: 0.2022 | |
| Epoch: 5 | |
| Training Loss: 0.1920 | |
| Epoch: 6 | |
| Training Loss: 0.1829 | |
| Epoch: 7 | |
| Training Loss: 0.1617 | |
| Epoch: 8 | |
| Training Loss: 0.1658 | |
| Epoch: 9 | |
| Training Loss: 0.1693 | |
| Epoch: 10 | |
| Training Loss: 0.1623 | |
| Epoch: 1 | |
| Training Loss: 0.6639 | |
| Epoch: 2 | |
| Training Loss: 0.4886 | |
| Epoch: 3 | |
| Training Loss: 0.4225 | |
| Epoch: 4 | |
| Training Loss: 0.4258 | |
| Epoch: 5 | |
| Training Loss: 0.3178 | |
| Epoch: 6 | |
| Training Loss: 0.3119 | |
| Epoch: 7 | |
| Training Loss: 0.4068 | |
| Epoch: 8 | |
| Training Loss: 0.3462 | |
| Epoch: 9 | |
| Training Loss: 0.3247 | |
| Epoch: 10 | |
| Training Loss: 0.5077 | |
| Epoch: 11 | |
| Training Loss: 0.5386 | |
| Epoch: 12 | |
| Training Loss: 0.3684 | |
| Epoch: 13 | |
| Training Loss: 0.4887 | |
| Epoch: 14 | |
| Training Loss: 0.4378 | |
| Epoch: 15 | |
| Training Loss: 0.3200 | |
| Epoch: 16 | |
| Training Loss: 0.4941 | |
| Epoch: 17 | |
| Training Loss: 0.1979 | |
| Epoch: 18 | |
| Training Loss: 0.4333 | |
| Epoch: 19 | |
| Training Loss: 0.1584 | |
| Epoch: 20 | |
| Training Loss: 0.4017 | |
| Epoch: 90 | |
| Training Loss: 0.1385 | |
| Epoch: 91 | |
| Training Loss: 0.2615 | |
| Epoch: 92 | |
| Training Loss: 0.3430 | |
| Epoch: 93 | |
| Training Loss: 0.2878 | |
| Epoch: 94 | |
| Training Loss: 0.1983 | |
| Epoch: 95 | |
| Training Loss: 0.4732 | |
| Epoch: 96 | |
| Training Loss: 0.2313 | |
| Epoch: 97 | |
| Training Loss: 0.2635 | |
| Epoch: 98 | |
| Training Loss: 0.3285 | |
| Epoch: 99 | |
| Training Loss: 0.3422 | |
| Epoch: 100 | |
| Training Loss: 0.2398 | |
| Test Accuracy is: 84.59% | Test Accuracy is: 88.53% |

Appendix for problem 3

1. Please download the provided code, 'autoencoder_sample.py'. We have provided a sketch code of a sample autoencoder network structure. Please fill in your work and run the training. Detailed comments can be found in the file. Note: this is not an optimal structure for this problem, you are encouraged to do more testing and tweaking. Improvement of structure(for Q3(a)-(c)) with analysis will receive bonus points, but it is not required.
* Make sure you have saved your model after your training (the same for Q3-(b)), because you will need to use this model in Q3-(c).
2. The sample code is using pytorch interface to download the Fashion MNIST dataset. You can either modify the input based on this method, or write your own data loader. Please state your method in your report.
3. Please refer to the comments in the sample code to load your trained model and freeze layers. You need to figure out how to connect the fully-connected layer, and modify the training section to make it work.