# Machine Learning Homework 4_Report

**2017112167 이수진**

1. CNN & MNIST

**<u><full program code> : I wrote only the red part for each question.</u>**

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.utils.data as Data
import torchvision
import matplotlib.pyplot as plt
#matplotlib inline

torch.manual_seed(1)    # reproducible
# Hyper Parameters
EPOCH = 1               # train the training data n times, to save time, we just train 1
epoch
BATCH_SIZE = 50
LR = 0.001              # learning rate
DOWNLOAD_MNIST = False   # set to False if you have downloaded

# Mnist digits dataset
train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,                                     # this is training data
    transform=torchvision.transforms.ToTensor(),    # Converts a PIL.Image or numpy.ndarray t
o
                                                    # torch.FloatTensor of shape (C x
H x W) and normalize in the range [0.0, 1.0]
    download=False,                        # download it if you don't have it
)
# plot one example
print(train_data.train_data.size())                 # (60000, 28, 28)
print(train_data.train_labels.size())               # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()

# Data Loader for easy mini-batch return in training, the image batch shape will be (50, 1,
28, 28)
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

# convert test data into Variable, pick 2000 samples to speed up testing
test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)
print(test_data.test_data[0].size())
test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1)).type(torch.FloatTensor)[:2000]/255
.
print(test_x[0].size())
# shape from (2000, 28, 28) to (2000, 1, 28, 28), value in range(0,1)
test_y = test_data.test_labels[:2000]
print(test_y[0])

class CNN(nn.Module):
    def __init__(self):
```

```
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=5,              # filter size
                stride=1,                   # filter movement/step
                padding=2,                  # if want same width and length of this im
age after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape
 (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32
* 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotte
d

# following function (plot_with_labels) is for visualization, can be ignored if not interested
from matplotlib import cm
try: from sklearn.manifold import TSNE; HAS_SK = True
except: HAS_SK = False; print('Please install sklearn for layer visualization')
def plot_with_labels(lowDWeights, labels):
    plt.cla()
    X, Y = lowDWeights[:, 0], lowDWeights[:, 1]
    for x, y, s in zip(X, Y, labels):
        c = cm.rainbow(int(255 * s / 9)); plt.text(x, y, s, backgroundcolor=c, fontsize=9)
    plt.xlim(X.min(), X.max()); plt.ylim(Y.min(), Y.max()); plt.title('Visualize last layer'); plt.show();
 plt.pause(0.01)

plt.ion()
# training and testing
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):   # gives batch data, normalize x when itera
te train_loader
        b_x = Variable(x)   # batch x
        b_y = Variable(y)   # batch y

        output = cnn(b_x)[0]               # cnn output
```

```
            loss = loss_func(output, b_y)     # cross entropy loss
            optimizer.zero_grad()             # clear gradients for this training step
            loss.backward()                   # backpropagation, compute gradients
            optimizer.step()                  # apply gradients

            if step % 100 == 0:
                test_output, last_layer = cnn(test_x)
                pred_y = torch.max(test_output, 1)[1].data.squeeze()
                accuracy = (pred_y == test_y).sum().item() / float(test_y.size(0))
                print('Epoch: ', epoch, '| train loss: %.4f' % loss.data, '| test accuracy: %.2f' % 
accuracy)
                if HAS_SK:
                    # Visualization of trained flatten layer (T-SNE)
                    tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
                    plot_only = 500
                    low_dim_embs = tsne.fit_transform(last_layer.data.numpy()[:plot_only, :])
                    labels = test_y.numpy()[:plot_only]
                    plot_with_labels(low_dim_embs, labels)
plt.ioff()
```

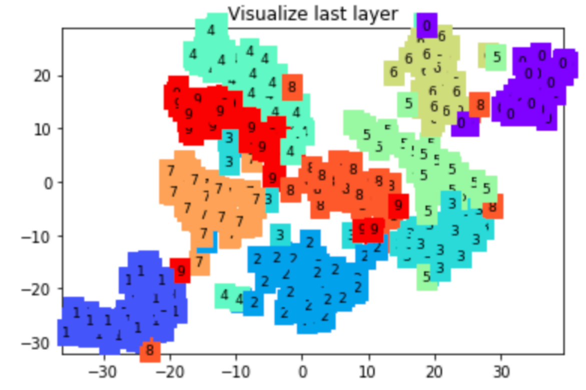
2) change the current kernel size of the program to different size. (Change ‘kernel_size’ parameter of ‘Conv2D’ function.) Repeat this three times and compare the results.

**#1-2 kernel_size=7**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=7,              # filter size
                stride=1,                   # filter movement/step
                padding=3,                  # if want same width and length of this image 
after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 
14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 7, 1, 3),     # output shape (32, 14, 14)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization
cnn = CNN()
print(cnn)  # net architecture
```

```
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)     # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                           # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.0769 | test accuracy: 0.98
```

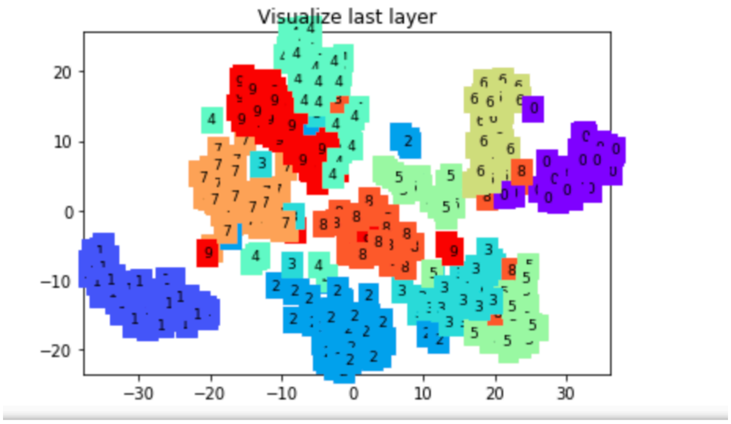



**#1-2 kernel_size= 5**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(          # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,               # input height
                out_channels=16,             # n_filters
                kernel_size=5,               # filter size
                stride=1,                    # filter movement/step
                padding=2,                   # if want same width and length of this image
after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                               # output shape (16, 28, 28)
            nn.ReLU(),                       # activation
            nn.MaxPool2d(kernel_size=2),     # choose max value in 2x2 area, output shape (16,
14, 14)
        )
        self.conv2 = nn.Sequential(          # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),      # output shape (32, 14, 14)
            nn.ReLU(),                       # activation
            nn.MaxPool2d(2),                 # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)    # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 *
7)
        output = self.out(x)
        return output, x    # return x for visualization
cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)     # optimize all cnn parameters
```

```
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

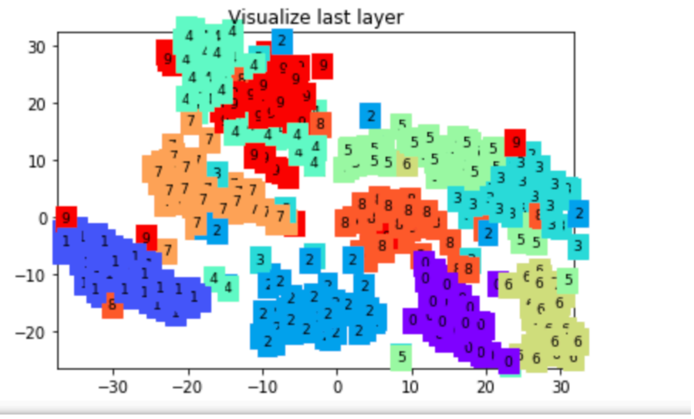
```
Epoch:  0 | train loss: 0.0257 | test accuracy: 0.98
```




**#1-2 kernel_size= 3**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=3,              # filter size
                stride=1,                   # filter movement/step
                padding=1,                  # if want same width and length of this image after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 3, 1, 1),     # output shape (32, 14, 14)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization
cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.1776 | test accuracy: 0.97
```



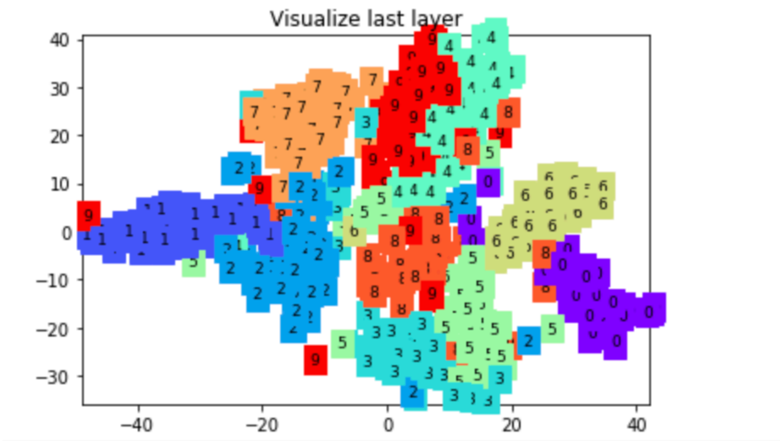

**Report**

⇨ **When kernel_size is 7, 5, or 5, the comparison was made. Among the results with the highest test accuracy, the graph with the lowest train loss was selected and compared. Accuracy increased as kernel_size increased, and train loss increased as kernel_size decreased.**

3) remove pooling layer in the program (you can remove 'MaxPool2D' function) and compare the results

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),

        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),

        )
        self.out = nn.Linear(32 * 28 * 28, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x      # return x for visualization
cnn = CNN()
print(cnn)    # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)     # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                         # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.0493 | test accuracy: 0.98
```




**<u>Report</u>**

⇨ **<u>The accuracy is similar, but the train loss increases.</u>**

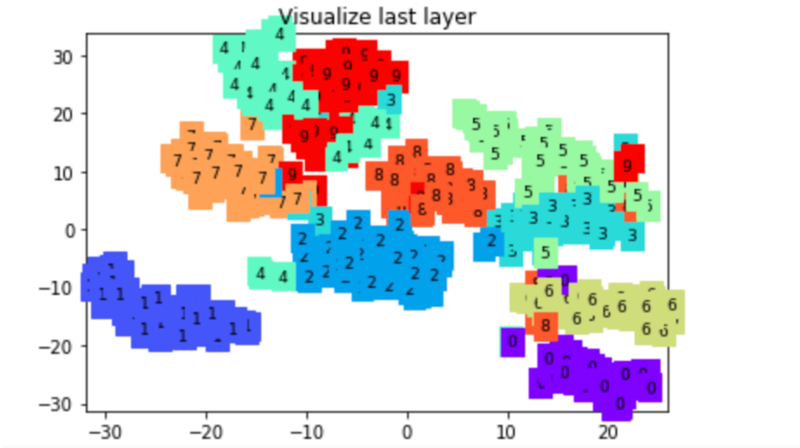
4) change the current activation function to other activation function (e.g. sigmoid, tanh, etc). You can do so by nn.Sigmoid() to nn.ReLU(), nn.Tanh(), etc) Repeat this three times and compare the results.

**#1-4 ReLU()**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=5,              # filter size
                stride=1,                   # filter movement/step
                padding=2,                  # if want same width and length of this image after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
```

```
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization
cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.0257 | test accuracy: 0.98
```

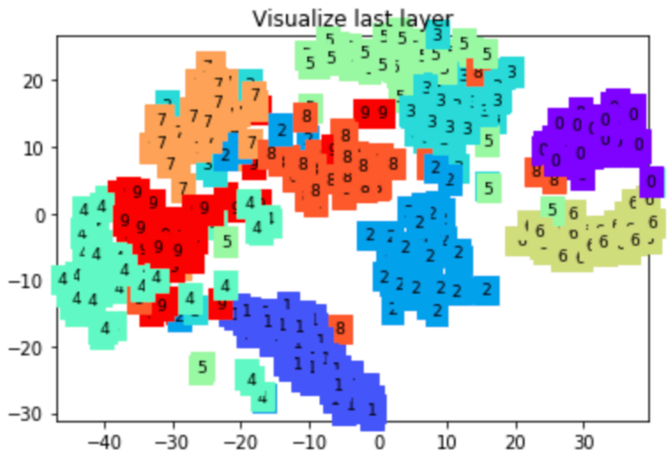



**#1-4 Tanh()**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=5,              # filter size
                stride=1,                   # filter movement/step
                padding=2,                  # if want same width and length of this image after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.Tanh(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
            nn.Tanh(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x     # return x for visualization
cnn = CNN()
print(cnn)   # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)    # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                        # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.0224 | test accuracy: 0.98
```

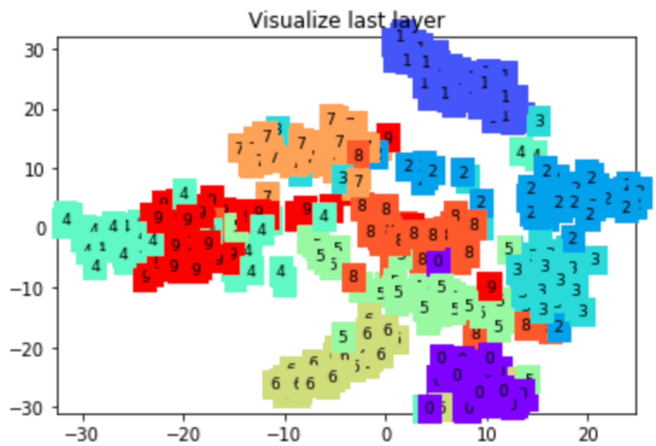



**#1-4 Sigmoid()**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(            # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,                 # input height
                out_channels=16,               # n_filters
                kernel_size=5,                 # filter size
                stride=1,                      # filter movement/step
                padding=2,                     # if want same width and length of this image
after con2d, padding=(kernel_size-1)/2 if stride=1
            ),                                 # output shape (16, 28, 28)
            nn.Sigmoid(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16,
14, 14)
        )
        self.conv2 = nn.Sequential(           # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),       # output shape (32, 14, 14)
```

```
            nn.Sigmoid(),                         # activation
            nn.MaxPool2d(2),                      # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)    # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)               # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x     # return x for visualization
cnn = CNN()
print(cnn)    # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)    # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                         # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.1897 | test accuracy: 0.94
```




**Report**

⇨ **When using Tanh() and ReLU(), the accuracy is higher and the train loss is smaller than when using Sigmoid(). Sigmoid() has poor function.**

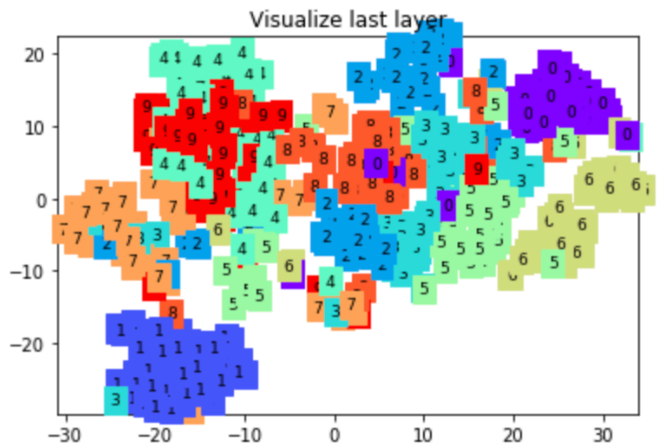
5) change the current optimization method to other optimization methods (e.g. adam, adaGrad, RMSProp, adaDelta, etc). You can use torch.optim.Adam, etc. Repeat this three times and compare the results.

**#1-5 Adagrad**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(          # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,               # input height
                out_channels=16,             # n_filters
                kernel_size=5,               # filter size
                stride=1,                    # filter movement/step
                padding=2,                   # if want same width and length of this image after con2d,
```

```
padding=(kernel_size-1)/2 if stride=1
            ),                                  # output shape (16, 28, 28)
            nn.ReLU(),                          # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
            nn.ReLU(),                          # activation
            nn.MaxPool2d(2),                    # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adagrad(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.2532 | test accuracy: 0.88
```

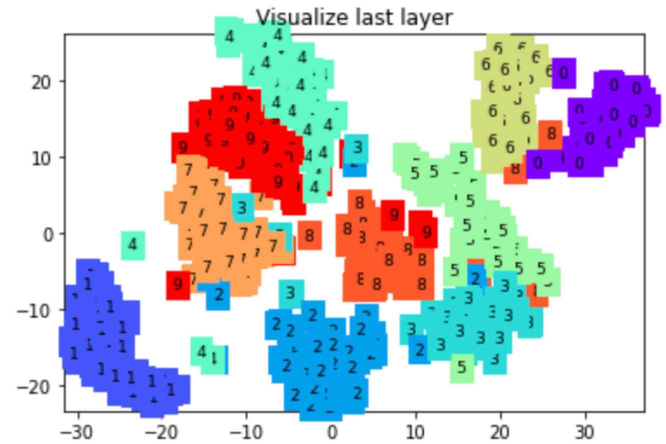



**#1-5 RMSprop**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=5,              # filter size
                stride=1,                   # filter movement/step
                padding=2,                  # if want same width and length of this image after con2d,
padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
```

```
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),     # output shape (32, 14, 14)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(2),                # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)   # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.RMSprop(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()
```

```
Epoch:  0 | train loss: 0.0280 | test accuracy: 0.98
```

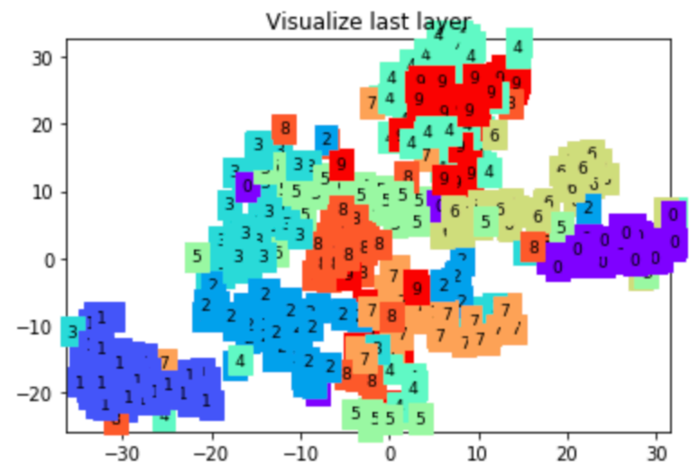



#1-5 Adadelta

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(         # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,              # input height
                out_channels=16,            # n_filters
                kernel_size=5,              # filter size
                stride=1,                   # filter movement/step
                padding=2,                  # if want same width and length of this image after con2d,
padding=(kernel_size-1)/2 if stride=1
            ),                              # output shape (16, 28, 28)
            nn.ReLU(),                      # activation
            nn.MaxPool2d(kernel_size=2),    # choose max value in 2x2 area, output shape (16, 14, 14)
        )
```

```
        self.conv2 = nn.Sequential(          # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),      # output shape (32, 14, 14)
            nn.ReLU(),                       # activation
            nn.MaxPool2d(2),                 # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)    # fully connected layer, output 10 classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)          # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x     # return x for visualization

cnn = CNN()
print(cnn)   # net architecture

optimizer = torch.optim.Adadelta(cnn.parameters(), lr=LR)    # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()
```

```
Epoch:  0 | train loss: 2.2464 | test accuracy: 0.34
```




**Report**

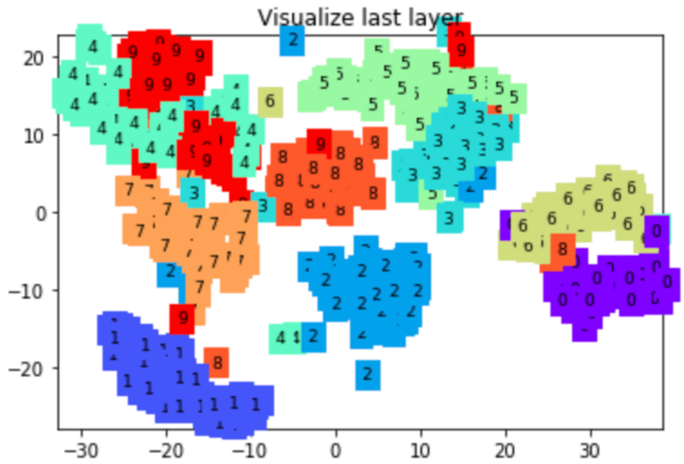
⇨ **Adadelta's accuracy is very poor. Adagrad is more accurate than Adadelta, but not as good. RMSprop is much more accurate than others and the train loss is very small.**

6) now add the Xavier weight initialization method and compare the results. (use torch.nn.init.xavier_uniform)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(          # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,               # input height
                out_channels=16,             # n_filters
                kernel_size=5,               # filter size
                stride=1,                    # filter movement/step
                padding=2,                   # if want same width and length of this image after con2d,
padding=(kernel_size-1)/2 if stride=1
            ),                               # output shape (16, 28, 28)
            nn.ReLU(),                       # activation
```

```
            nn.MaxPool2d(kernel_size=2),       # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(             # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),        # output shape (32, 14, 14)
            nn.ReLU(),                          # activation
            nn.MaxPool2d(2),                    # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)    # fully connected layer, output 10 classes
        torch.nn.init.xavier_uniform_(self.out.weight)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)               # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)   # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)    # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                        # the target label is not one-hotted
```

```
Epoch:  0 | train loss: 0.1222 | test accuracy: 0.98
```




**Report**

⇨ **The accuracy is similar, but the train loss increases.**

7) choose ONE other parameters of CNN program (e.g. number of hidden nodes, dropout, epochs, batch normalization, etc). Change the value of this parameter and compare the results.

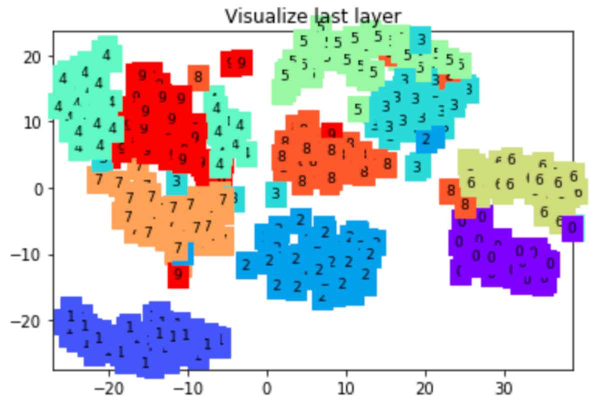
**#1-7 EPOCH = 2**

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.utils.data as Data
import torchvision
import matplotlib.pyplot as plt
#matplotlib inline
```

```
torch.manual_seed(1)       # reproducible
# Hyper Parameters
EPOCH = 2                        # train the training data n times, to save time, we just train 1 epoch
BATCH_SIZE = 50
LR = 0.001                       # learning rate
DOWNLOAD_MNIST = False      # set to False if you have downloaded

# Mnist digits dataset
train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,                                                  # this is training data
    transform=torchvision.transforms.ToTensor(),       # Converts a PIL.Image or numpy.ndarray to
                                                                   # torch.FloatTensor of shape (C x H x W) and
normalize in the range [0.0, 1.0]
    download=False,                                    # download it if you don't have it
)
# plot one example
print(train_data.train_data.size())                          # (60000, 28, 28)
print(train_data.train_labels.size())                       # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()

# Data Loader for easy mini-batch return in training, the image batch shape will be (50, 1, 28, 28)
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

# convert test data into Variable, pick 2000 samples to speed up testing
test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)
print(test_data.test_data[0].size())
test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1)).type(torch.FloatTensor)[:2000]/255.
print(test_x[0].size())
# shape from (2000, 28, 28) to (2000, 1, 28, 28), value in range(0,1)
test_y = test_data.test_labels[:2000]
print(test_y[0])

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(                # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,                        # input height
                out_channels=16,                     # n_filters
                kernel_size=5,                        # filter size
                stride=1,                              # filter movement/step
                padding=2,                             # if want same width and length of this image after con2d,
padding=(kernel_size-1)/2 if stride=1
            ),                                            # output shape (16, 28, 28)
            nn.ReLU(),                                 # activation
            nn.MaxPool2d(kernel_size=2),       # choose max value in 2x2 area, output shape (16, 14, 14)
        )
        self.conv2 = nn.Sequential(                # input shape (1, 28, 28)
            nn.Conv2d(16, 32, 5, 1, 2),         # output shape (32, 14, 14)
            nn.ReLU(),                                 # activation
            nn.MaxPool2d(2),                         # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 10)      # fully connected layer, output 10 classes
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)           # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)  # net architecture

optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted

# following function (plot_with_labels) is for visualization, can be ignored if not interested
from matplotlib import cm
try: from sklearn.manifold import TSNE; HAS_SK = True
except: HAS_SK = False; print('Please install sklearn for layer visualization')
def plot_with_labels(lowDWeights, labels):
    plt.cla()
    X, Y = lowDWeights[:, 0], lowDWeights[:, 1]
    for x, y, s in zip(X, Y, labels):
        c = cm.rainbow(int(255 * s / 9)); plt.text(x, y, s, backgroundcolor=c, fontsize=9)
    plt.xlim(X.min(), X.max()); plt.ylim(Y.min(), Y.max()); plt.title('Visualize last layer'); plt.show(); plt.pause(0.01)

plt.ion()
# training and testing
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):   # gives batch data, normalize x when iterate train_loader
        b_x = Variable(x)   # batch x
        b_y = Variable(y)   # batch y

        output = cnn(b_x)[0]               # cnn output
        loss = loss_func(output, b_y)   # cross entropy loss
        optimizer.zero_grad()           # clear gradients for this training step
        loss.backward()                 # backpropagation, compute gradients
        optimizer.step()                # apply gradients

        if step % 100 == 0:
            test_output, last_layer = cnn(test_x)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == test_y).sum().item() / float(test_y.size(0))
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data, '| test accuracy: %.2f' % accuracy)
            if HAS_SK:
                # Visualization of trained flatten layer (T-SNE)
                tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
                plot_only = 500
                low_dim_embs = tsne.fit_transform(last_layer.data.numpy()[:plot_only, :])
                labels = test_y.numpy()[:plot_only]
                plot_with_labels(low_dim_embs, labels)
plt.ioff()
```

```
Epoch:  1 | train loss: 0.0023 | test accuracy: 0.99
```




**Report**

⇨ **I changed EPOCH as 2. The test accuracy is increased, and train loss decreased. This is the best result.**

2. CNN & CIFAR-10

**<full program code> : I wrote only the red part for each question.**

```
import torch
import torchvision
import torchvision.transforms as transforms


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

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

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        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 = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
net.to(device)

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

print('start traning.')
for epoch in range(2):    # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:        # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
```

```
print('Finished Training')

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

# del dataiter
```

2) kernal size

```
#2-2 kernal size = 5
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        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 = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x


net = Net()


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
net.to(device)

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```
cpu
start traning.
[1,  2000] loss: 2.165
[1,  4000] loss: 1.801
[1,  6000] loss: 1.598
[1,  8000] loss: 1.514
[1, 10000] loss: 1.429
[1, 12000] loss: 1.389
[2,  2000] loss: 1.324
[2,  4000] loss: 1.295
[2,  6000] loss: 1.270
[2,  8000] loss: 1.234
[2, 10000] loss: 1.214
[2, 12000] loss: 1.183
Finished Training
Accuracy of the network on the 10000 test images: 59 %
Accuracy of plane : 60 %
Accuracy of   car : 79 %
Accuracy of  bird : 36 %
Accuracy of   cat : 26 %
Accuracy of  deer : 62 %
Accuracy of   dog : 55 %
Accuracy of  frog : 69 %
Accuracy of horse : 70 %
Accuracy of  ship : 72 %
Accuracy of truck : 57 %
```

```
#2-2 kernal size = 7
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 7)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 16, 7)
        self.fc1 = nn.Linear(16 * 3 * 3, 120)
        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 = x.view(-1, 16 * 3 * 3)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x

net = Net()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
net.to(device)

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```
cpu
start traning.
[1,  2000] loss: 2.182
[1,  4000] loss: 1.817
[1,  6000] loss: 1.688
[1,  8000] loss: 1.588
[1, 10000] loss: 1.517
[1, 12000] loss: 1.486
[2,  2000] loss: 1.413
[2,  4000] loss: 1.419
[2,  6000] loss: 1.349
[2,  8000] loss: 1.352
[2, 10000] loss: 1.346
[2, 12000] loss: 1.292
Finished Training
Accuracy of the network on the 10000 test images: 54 %
Accuracy of plane : 51 %
Accuracy of   car : 72 %
Accuracy of  bird : 44 %
Accuracy of   cat : 31 %
Accuracy of  deer : 33 %
Accuracy of   dog : 42 %
Accuracy of  frog : 73 %
Accuracy of horse : 64 %
Accuracy of  ship : 62 %
Accuracy of truck : 65 %
```

**Report**

> **Kernal size 7 is better accuracy than 5**

3. RNN & Mnist

**<full program code> : I wrote only the red part for each question.**

```
import torch
from torch import nn
from torch.autograd import Variable
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

torch.manual_seed(1)    # reproducible

# Hyper Parameters
EPOCH = 1               # train the training data n times, to save time, we just train 1 epoch
BATCH_SIZE = 64
TIME_STEP = 28          # rnn time step / image height
```

```
INPUT_SIZE = 28              # rnn input size / image width
LR = 0.01                    # learning rate
DOWNLOAD_MNIST = True     # set to True if haven't download the data

# Mnist digital dataset
train_data = dsets.MNIST(
    root='./mnist/',
    train=True,                                    # this is training data
    transform=transforms.ToTensor(),      # Converts a PIL.Image or numpy.ndarray to
                                                   # torch.FloatTensor of shape (C x H x W) and normalize in the
range [0.0, 1.0]
    download=DOWNLOAD_MNIST,                 # download it if you don't have it
)

# plot one example
print(train_data.train_data.size())       # (60000, 28, 28)
print(train_data.train_labels.size())     # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()

# Data Loader for easy mini-batch return in training
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

# convert test data into Variable, pick 2000 samples to speed up testing
test_data = dsets.MNIST(root='./mnist/', train=False, transform=transforms.ToTensor())
test_x = Variable(test_data.test_data, volatile=True).type(torch.FloatTensor)[:2000]/255.    # shape (2000, 28,
28) value in range(0,1)
test_y = test_data.test_labels.numpy().squeeze()[:2000]     # covert to numpy array

class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(            # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,           # rnn hidden unit
            num_layers=1,             # number of rnn layer
            batch_first=True,         # input & output will has batch size as 1s dimension. e.g. (batch,
time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)    # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)
```

```
optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                           # the target label is not one-hotted

# training and testing
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):          # gives batch data
        b_x = Variable(x.view(-1, 28, 28))                # reshape x to (batch, time_step, input_size)
        b_y = Variable(y)                                 # batch y

        output = rnn(b_x)                                 # rnn output
        loss = loss_func(output, b_y)                     # cross entropy loss
        optimizer.zero_grad()                             # clear gradients for this training step
        loss.backward()                                   # backpropagation, compute gradients
        optimizer.step()                                  # apply gradients

        if step % 50 == 0:
            test_output = rnn(test_x)                     # (samples, time_step, input_size)
            pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
            accuracy = sum(pred_y == test_y) / float(test_y.size)
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data, '| test accuracy: %.2f' % accuracy)
```

2) change the number of hidden nodes in the program three times and compare the results.

```
#3-2 hidden_size=64

class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(           # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,           # rnn hidden unit
            num_layers=1,             # number of rnn layer
            batch_first=True,         # input & output will has batch size as 1s dimension. e.g. (batch,
time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)    # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)
```

```
optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                           # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 64, batch_first=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
Epoch:  0 | train loss: 2.2883 | test accuracy: 0.10
Epoch:  0 | train loss: 0.8980 | test accuracy: 0.56
Epoch:  0 | train loss: 1.0743 | test accuracy: 0.70
Epoch:  0 | train loss: 0.6517 | test accuracy: 0.83
Epoch:  0 | train loss: 0.5668 | test accuracy: 0.83
Epoch:  0 | train loss: 0.3297 | test accuracy: 0.88
Epoch:  0 | train loss: 0.4544 | test accuracy: 0.89
Epoch:  0 | train loss: 0.3315 | test accuracy: 0.92
Epoch:  0 | train loss: 0.1421 | test accuracy: 0.92
Epoch:  0 | train loss: 0.3268 | test accuracy: 0.93
Epoch:  0 | train loss: 0.0576 | test accuracy: 0.93
Epoch:  0 | train loss: 0.2015 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1035 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1204 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1826 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1199 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0627 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1477 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2381 | test accuracy: 0.95
```

```
#3-2 hidden_size=32

class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=32,         # rnn hidden unit
            num_layers=1,           # number of rnn layer
            batch_first=True,       # input & output will has batch size as 1s dimension. e.g. (batch, time_step, input_size)
        )

        self.out = nn.Linear(32, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                           # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 32, batch_first=True)
  (out): Linear(in_features=32, out_features=10, bias=True)
)
```

```
/Users/soojinlee/opt/anaconda3/lib/python3.7/site-packages/ip
d and now has no effect. Use `with torch.no_grad():` instead.
```

```
Epoch:  0 | train loss: 2.3050 | test accuracy: 0.12
Epoch:  0 | train loss: 1.3808 | test accuracy: 0.49
Epoch:  0 | train loss: 1.0123 | test accuracy: 0.66
Epoch:  0 | train loss: 0.6900 | test accuracy: 0.77
Epoch:  0 | train loss: 0.7815 | test accuracy: 0.79
Epoch:  0 | train loss: 0.3825 | test accuracy: 0.81
Epoch:  0 | train loss: 0.5824 | test accuracy: 0.85
Epoch:  0 | train loss: 0.3229 | test accuracy: 0.89
Epoch:  0 | train loss: 0.1642 | test accuracy: 0.89
Epoch:  0 | train loss: 0.3344 | test accuracy: 0.90
Epoch:  0 | train loss: 0.4474 | test accuracy: 0.92
Epoch:  0 | train loss: 0.2098 | test accuracy: 0.93
Epoch:  0 | train loss: 0.5863 | test accuracy: 0.92
Epoch:  0 | train loss: 0.2316 | test accuracy: 0.92
Epoch:  0 | train loss: 0.0935 | test accuracy: 0.94
Epoch:  0 | train loss: 0.2247 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1834 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1711 | test accuracy: 0.94
Epoch:  0 | train loss: 0.3964 | test accuracy: 0.95
```

```
#3-2 hidden_size=16

class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=16,         # rnn hidden unit
            num_layers=1,           # number of rnn layer
            batch_first=True,       # input & output will has batch size as 1s dimension. e.g. (batch, time_step, input_size)
        )

        self.out = nn.Linear(16, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                       # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 16, batch_first=True)
  (out): Linear(in_features=16, out_features=10, bias=True)
)
```

```
/Users/soojinlee/opt/anaconda3/lib/python3.7/site-packages/ip
d and now has no effect. Use `with torch.no_grad():` instead.
```

```
Epoch:  0 | train loss: 2.3317 | test accuracy: 0.11
Epoch:  0 | train loss: 1.3542 | test accuracy: 0.45
Epoch:  0 | train loss: 1.0651 | test accuracy: 0.54
Epoch:  0 | train loss: 1.0748 | test accuracy: 0.60
Epoch:  0 | train loss: 0.6775 | test accuracy: 0.70
Epoch:  0 | train loss: 0.7007 | test accuracy: 0.74
Epoch:  0 | train loss: 0.6877 | test accuracy: 0.71
Epoch:  0 | train loss: 0.6917 | test accuracy: 0.79
Epoch:  0 | train loss: 0.3936 | test accuracy: 0.82
Epoch:  0 | train loss: 0.3910 | test accuracy: 0.83
Epoch:  0 | train loss: 0.4074 | test accuracy: 0.85
Epoch:  0 | train loss: 0.3370 | test accuracy: 0.86
Epoch:  0 | train loss: 0.3332 | test accuracy: 0.86
Epoch:  0 | train loss: 0.3917 | test accuracy: 0.88
Epoch:  0 | train loss: 0.3024 | test accuracy: 0.88
Epoch:  0 | train loss: 0.4465 | test accuracy: 0.89
Epoch:  0 | train loss: 0.3496 | test accuracy: 0.88
Epoch:  0 | train loss: 0.3620 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2999 | test accuracy: 0.88
```

**Report**

⇨ **The smaller the hidden size, the lower the accuracy.**

3) change the current optimization method to other optimization methods

#3-3 Adam

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,         # rnn hidden unit
            num_layers=1,           # number of rnn layer
            batch_first=True,       # input & output will has batch size as 1s dimension. e.g. (batch, time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out
```

```
rnn = RNN()
print(rnn)

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                          # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 64, batch_first=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
Epoch:  0 | train loss: 2.2883 | test accuracy: 0.10
Epoch:  0 | train loss: 0.8980 | test accuracy: 0.56
Epoch:  0 | train loss: 1.0743 | test accuracy: 0.70
Epoch:  0 | train loss: 0.6517 | test accuracy: 0.83
Epoch:  0 | train loss: 0.5668 | test accuracy: 0.83
Epoch:  0 | train loss: 0.3297 | test accuracy: 0.88
Epoch:  0 | train loss: 0.4544 | test accuracy: 0.89
Epoch:  0 | train loss: 0.3315 | test accuracy: 0.92
Epoch:  0 | train loss: 0.1421 | test accuracy: 0.92
Epoch:  0 | train loss: 0.3268 | test accuracy: 0.93
Epoch:  0 | train loss: 0.0576 | test accuracy: 0.93
Epoch:  0 | train loss: 0.2015 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1035 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1204 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1826 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1199 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0627 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1477 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2381 | test accuracy: 0.95
```

#3-3 optimizer – RMSprop

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,         # rnn hidden unit
            num_layers=1,           # number of rnn layer
            batch_first=True,       # input & output will has batch size as 1s dimension. e.g. (batch, time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)
```

```
optimizer = torch.optim.RMSprop(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                              # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 64, batch_first=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
/Users/soojinlee/opt/anaconda3/lib/python3.7/site-packages/ipy
d and now has no effect. Use `with torch.no_grad():` instead.
Epoch:  0 | train loss: 2.2883 | test accuracy: 0.10
Epoch:  0 | train loss: 1.2381 | test accuracy: 0.43
Epoch:  0 | train loss: 0.8018 | test accuracy: 0.64
Epoch:  0 | train loss: 0.7949 | test accuracy: 0.74
Epoch:  0 | train loss: 0.4477 | test accuracy: 0.78
Epoch:  0 | train loss: 0.3670 | test accuracy: 0.84
Epoch:  0 | train loss: 0.4815 | test accuracy: 0.87
Epoch:  0 | train loss: 0.5720 | test accuracy: 0.85
Epoch:  0 | train loss: 0.3554 | test accuracy: 0.89
Epoch:  0 | train loss: 0.3389 | test accuracy: 0.90
Epoch:  0 | train loss: 0.1689 | test accuracy: 0.92
Epoch:  0 | train loss: 0.2035 | test accuracy: 0.91
Epoch:  0 | train loss: 0.1487 | test accuracy: 0.93
Epoch:  0 | train loss: 0.2166 | test accuracy: 0.92
Epoch:  0 | train loss: 0.2518 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1977 | test accuracy: 0.92
Epoch:  0 | train loss: 0.2616 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1658 | test accuracy: 0.94
Epoch:  0 | train loss: 0.3873 | test accuracy: 0.94
```

#3-3 optimizer – Adadelta

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(         # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,         # rnn hidden unit
            num_layers=1,           # number of rnn layer
            batch_first=True,       # input & output will has batch size as 1s dimension. e.g. (batch, time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)   # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out
```

```
rnn = RNN()
print(rnn)

optimizer = torch.optim.Adadelta(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                               # the target label is not one-hotted
```

```
RNN(
  (rnn): LSTM(28, 64, batch_first=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
/Users/soojinlee/opt/anaconda3/lib/python3.7/site-packages/ipy
d and now has no effect. Use `with torch.no_grad():` instead.
Epoch:  0 | train loss: 2.2883 | test accuracy: 0.08
Epoch:  0 | train loss: 2.3065 | test accuracy: 0.08
Epoch:  0 | train loss: 2.3140 | test accuracy: 0.09
Epoch:  0 | train loss: 2.3099 | test accuracy: 0.09
Epoch:  0 | train loss: 2.3207 | test accuracy: 0.09
Epoch:  0 | train loss: 2.2916 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3055 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3028 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3026 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3081 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3226 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3133 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3073 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3168 | test accuracy: 0.10
Epoch:  0 | train loss: 2.2932 | test accuracy: 0.10
Epoch:  0 | train loss: 2.2968 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3007 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3038 | test accuracy: 0.10
Epoch:  0 | train loss: 2.3010 | test accuracy: 0.10
```

**Report**

- ⇨ **adadelta cannot be used because it is very inaccurate to use. rmsprop is also good accuracy, but adam's accuracy is the best.**

4) change LSTM to GRU (or vice versa). Compare the results

```
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.GRU(          # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,          # rnn hidden unit
            num_layers=2,            # number of rnn layer
            batch_first=True,        # input & output will has batch size as 1s dimension. e.g. (batch,
time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)     # None represents zero initial hidden state
```

```
        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)

optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)    # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                          # the target label is not one-hotted
```

```
RNN(
  (rnn): GRU(28, 64, num_layers=2, batch_first=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
)
/Users/soojinlee/opt/anaconda3/lib/python3.7/site-packages/i
d and now has no effect. Use `with torch.no_grad():` instead
Epoch:  0 | train loss: 2.3132 | test accuracy: 0.12
Epoch:  0 | train loss: 0.8244 | test accuracy: 0.67
Epoch:  0 | train loss: 0.3006 | test accuracy: 0.85
Epoch:  0 | train loss: 0.2121 | test accuracy: 0.90
Epoch:  0 | train loss: 0.1530 | test accuracy: 0.92
Epoch:  0 | train loss: 0.3483 | test accuracy: 0.92
Epoch:  0 | train loss: 0.0755 | test accuracy: 0.94
Epoch:  0 | train loss: 0.2027 | test accuracy: 0.95
Epoch:  0 | train loss: 0.3304 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0169 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2831 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1494 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2699 | test accuracy: 0.96
Epoch:  0 | train loss: 0.0510 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2100 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1083 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0791 | test accuracy: 0.97
Epoch:  0 | train loss: 0.0573 | test accuracy: 0.97
Epoch:  0 | train loss: 0.3135 | test accuracy: 0.96
```

**Report**

⇨ **GRU's accuracy is better than LSTM**

5) choose ONE other parameters of RNN program (e.g. batch_size, epochs, etc). Change the value of this parameter and compare the results.

#3-5 EPOCH=2

```
import torch
from torch import nn
from torch.autograd import Variable
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

torch.manual_seed(1)    # reproducible

# Hyper Parameters
EPOCH = 2               # train the training data n times, to save time, we just train 1 epoch
BATCH_SIZE = 64
TIME_STEP = 28          # rnn time step / image height
```

```
INPUT_SIZE = 28                 # rnn input size / image width
LR = 0.01                       # learning rate
DOWNLOAD_MNIST = False      # set to True if haven't download the data

# Mnist digital dataset
train_data = dsets.MNIST(
    root='./mnist/',
    train=True,                                     # this is training data
    transform=transforms.ToTensor(),        # Converts a PIL.Image or numpy.ndarray to
                                                    # torch.FloatTensor of shape (C x H x W) and normalize in the
range [0.0, 1.0]
    download=DOWNLOAD_MNIST,                        # download it if you don't have it
)

# plot one example
print(train_data.train_data.size())         # (60000, 28, 28)
print(train_data.train_labels.size())       # (60000)
plt.imshow(train_data.train_data[0].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[0])
plt.show()

# Data Loader for easy mini-batch return in training
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

# convert test data into Variable, pick 2000 samples to speed up testing
test_data = dsets.MNIST(root='./mnist/', train=False, transform=transforms.ToTensor())
test_x = Variable(test_data.test_data, volatile=True).type(torch.FloatTensor)[:2000]/255.      # shape (2000, 28,
28) value in range(0,1)
test_y = test_data.test_labels.numpy().squeeze()[:2000]       # covert to numpy array

class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()

        self.rnn = nn.LSTM(                 # if use nn.RNN(), it hardly learns
            input_size=INPUT_SIZE,
            hidden_size=64,                 # rnn hidden unit
            num_layers=1,                   # number of rnn layer
            batch_first=True,               # input & output will has batch size as 1s dimension. e.g. (batch,
time_step, input_size)
        )

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step, output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)      # None represents zero initial hidden state

        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out

rnn = RNN()
print(rnn)
```

```
optimizer = torch.optim.Adam(rnn.parameters(), lr=LR)      # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()                          # the target label is not one-hotted

# training and testing
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):           # gives batch data
        b_x = Variable(x.view(-1, 28, 28))                 # reshape x to (batch, time_step, input_size)
        b_y = Variable(y)                                  # batch y

        output = rnn(b_x)                                  # rnn output
        loss = loss_func(output, b_y)                      # cross entropy loss
        optimizer.zero_grad()                              # clear gradients for this training step
        loss.backward()                                    # backpropagation, compute gradients
        optimizer.step()                                   # apply gradients

        if step % 50 == 0:
            test_output = rnn(test_x)                      # (samples, time_step, input_size)
            pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
            accuracy = sum(pred_y == test_y) / float(test_y.size)
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data, '| test accuracy: %.2f' % accuracy)
```

```
Epoch:  0 | train loss: 0.2015 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1035 | test accuracy: 0.93
Epoch:  0 | train loss: 0.1204 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1826 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1199 | test accuracy: 0.95
Epoch:  0 | train loss: 0.0627 | test accuracy: 0.94
Epoch:  0 | train loss: 0.1477 | test accuracy: 0.96
Epoch:  0 | train loss: 0.2381 | test accuracy: 0.95
Epoch:  1 | train loss: 0.0900 | test accuracy: 0.95
Epoch:  1 | train loss: 0.0148 | test accuracy: 0.95
Epoch:  1 | train loss: 0.0904 | test accuracy: 0.95
Epoch:  1 | train loss: 0.1294 | test accuracy: 0.95
Epoch:  1 | train loss: 0.1356 | test accuracy: 0.96
Epoch:  1 | train loss: 0.1203 | test accuracy: 0.96
Epoch:  1 | train loss: 0.0942 | test accuracy: 0.96
Epoch:  1 | train loss: 0.1747 | test accuracy: 0.96
Epoch:  1 | train loss: 0.0608 | test accuracy: 0.96
Epoch:  1 | train loss: 0.1119 | test accuracy: 0.96
Epoch:  1 | train loss: 0.0941 | test accuracy: 0.96
Epoch:  1 | train loss: 0.0280 | test accuracy: 0.96
Epoch:  1 | train loss: 0.1034 | test accuracy: 0.96
Epoch:  1 | train loss: 0.2348 | test accuracy: 0.97
Epoch:  1 | train loss: 0.3485 | test accuracy: 0.96
Epoch:  1 | train loss: 0.1760 | test accuracy: 0.97
Epoch:  1 | train loss: 0.0346 | test accuracy: 0.97
Epoch:  1 | train loss: 0.0973 | test accuracy: 0.97
Epoch:  1 | train loss: 0.1013 | test accuracy: 0.97
```

**Report**

⇨ **By increase Epoch as 2, accuracy was increased. But running time is also increased.**

6) compare the accuracy of RNN for Mnist with that of CNN.

CNN's accuracy is higher than RNN on average. Also CNN's train loss is lower than RNN. CNN shows better function.