0. Import Dependencies

In [2]:

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

import pandas as pd
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

1. MNIST classification

main references:

```
Andrew Ng, Machine Learning, https://www.coursera.org/learn/machine-learning
https://zhuanlan.zhihu.com/p/38709373
https://github.com/alcarasj/simple-vision-stuff/blob/master/assignment2/main.py
https://github.com/bdura/humanware/blob/469c0a9a285b83f0951b6d88608613047894197f/models/CNN.
```

a. In the file mnist.py identify where the Stochastic Gradient Descent optimizer is created. Train the default CNN architecture by choosing appropriate parameter values for:

```
 i) Learning rate - explain what happens when you use too large or too small value and expl
ain why it is happening.

 ii) Weight decay - explain what happens when you use too large or too small regularization
and explain why it is happening.
```

Answer :

```
 i) In the process of  training, we can adjust the network weights by controlling the
updates rate. If the learning rate is too small, the loss function changes slowly and it wi
ll spend more time to converge, so it is easy to produce over-fitting; On the other hand, wh
en the learning rate is relatively large, the learning speed will be faster, but the gradie
nt decent can overshoot the minimum so it may fail to converge or even diverge. Therefore,
the value of the learning rate should be moderate.

 ii) In the loss function, weight decay is a coefficient placed in front of the
regularization. The regular term generally indicates the complexity of the model.
Therefore, the effect of weight decay is to adjust the influence of model complexity on the
loss function. The ultimate goal is to prevent Overfitting. If the weight decay is large, t
he value of the complex model loss function is large.
```

In [3]:

```
# Training hyperparameters
epochs = 1
batch_size = 64
learning_rate = 0.001# TODO
momentum = 0.9
weight_decay = 0.04 # TODO
log_interval = 20
```

In [4]:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
                               kernel_size=5, stride=1, padding=0)
        self.maxpool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
                               kernel_size=3, stride=1, padding=0)
        self.fc1 = nn.Linear(in_features=800, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=10)

        nn.init.kaiming_normal_(self.conv1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc2.weight, nonlinearity='linear')

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.maxpool(x)
        x = x.view(-1, 800)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

b) Correct the mistakes in CNN2 and train it on MNIST train set. Desired architecture of CNN2 is displayed on following diagam:

CONV KxK, N represents N features extracted by KxK filters, FC N represent fully-connected layer with N nodes. Set the padding so each convolution preserves input feature size.

In [4]:

```
class CNN2(nn.Module):
    def __init__(self):
        super(CNN2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32,
                               kernel_size=5, stride=1, padding=2)
        self.maxpool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
                               kernel_size=5, stride=1, padding=2)
        self.fc1 = nn.Linear(in_features=3136, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=10)

        nn.init.kaiming_normal_(self.conv1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.conv2.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.fc2.weight, nonlinearity='linear')

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.maxpool(x)
        x = x.view(-1, 3136)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

d) Design CNN3 with additional regularization of your choosing. Explain benefits of such regularization and report its accurracy on fashion MNIST and its relative improvement over CNN2.

Answer :

```
The accuracy rate has improved 3% compared with the previous one.
1) After the convolutional layer of the convolutional neural network, BatchNorm2d is  added
```

1) After the convolutional layer of the convolutional neural network, BatchNorm2d is added to normalize the data, which makes the data not unstable due to excessive data before the Relu is performed.
2) Modified the stride of the last layer of maxpool so that the downsampling is not twice as high.
3) Dropout reduces the dependencies between nodes, selectively ignoring individual neurons and avoiding overfitting. The entire dropout process is equivalent to averaging many different neural networks. Different networks produce different over-fittings, and some “reverse” fittings reduce the over-fitting as a whole. Besides, the dropout causes two neurons not necessarily to appear in one dropout network each time. Such updates of weights will no longer rely on the interaction of implicit nodes with fixed relationships, preventing situations where certain features are only effective under other specific features.

In [5]:

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

#         self.dropout = nn.Dropout(p=.05)

#         self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
#         self.bn1 = nn.BatchNorm2d(16)

#         self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
#         self.bn2 = nn.BatchNorm2d(32)

#         self.conv3 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
#         self.bn3 = nn.BatchNorm2d(32)

#         self.max = nn.MaxPool2d(2)

#         self.conv4 = nn.Conv2d(32, 64, kernel_size=3, padding=0)
#         self.bn4 = nn.BatchNorm2d(64)

#         self.fc1 = nn.Linear(in_features=9216, out_features=128)
#         self.fc2 = nn.Linear(in_features=128, out_features=10)

#         nn.init.kaiming_normal_(self.conv1.weight, nonlinearity='relu')
#         nn.init.kaiming_normal_(self.conv2.weight, nonlinearity='relu')
#         nn.init.kaiming_normal_(self.conv3.weight, nonlinearity='relu')
#         nn.init.kaiming_normal_(self.conv4.weight, nonlinearity='relu')
#         nn.init.kaiming_normal_(self.fc1.weight, nonlinearity='relu')
#         nn.init.kaiming_normal_(self.fc2.weight, nonlinearity='linear')

#     def forward(self, x):
#         n = x.size(0)

#         x = self.conv1(x)
#         x = self.bn1(x)
#         x = F.relu(x)
#         x = self.dropout(x)

#         x = self.conv2(x)
#         x = self.bn2(x)
#         x = F.relu(x)
#         x = self.dropout(x)

#         x = self.conv3(x)
#         x = self.bn3(x)
#         x = F.relu(x)
#         x = self.dropout(x)

#         x = self.max(x)
#         x = self.dropout(x)

#         x = self.conv4(x)
#         x = self.bn4(x)
#         x = F.relu(x)
#         x = self.dropout(x)

#         # Flatten the tensor for dense layers

#         x = x.view(-1,9216)
#         x = F.relu(x)
```

```
#         x = F.relu(x)
#         x = self.dropout(x)
#         x = self.fc1(x)
#         x = self.fc2(x)

#         return F.log_softmax(x, dim=1)
```

In [6]:

```
class CNN3(nn.Module):
    def __init__(self):
        super(CNN3, self).__init__()

        self.dropout = nn.Dropout(p=.05)

        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=1)
        )
        self.fc1 = nn.Linear(in_features=18432, out_features=10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = x.reshape(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc1(x)

        return F.log_softmax(x, dim=1)
```

In [5]:

```
# class CNN4(nn.Module):

#     def __init__(self):
#         super(CNN4, self).__init__()
#         self.layer1 = nn.Sequential(
#             nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
#             nn.BatchNorm2d(16),
#             nn.ReLU(),
#         )
#         self.layer2 = nn.Sequential(
#             nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
#             nn.BatchNorm2d(32),
#             nn.ReLU(),
#         )
#         self.layer3 = nn.Sequential(
#             nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
#             nn.BatchNorm2d(64),
#             nn.ReLU()
#         )
#         self.layer4 = nn.Sequential(
#             nn.Conv2d(64, 128, kernel_size=5, stride=1, padding=2),
#             nn.BatchNorm2d(128),
#             nn.ReLU(),
#             nn.MaxPool2d(kernel_size=3, stride=1)
```

```
#             nn.MaxPool2d(kernel_size=3, stride=1)
#         )
#         self.fc1 = nn.Linear(in_features=86528, out_features=10)

#     def forward(self, x):
#         x = self.layer1(x)
#         x = self.layer2(x)
#         x = self.layer3(x)
#         x = self.layer4(x)
#         x = x.reshape(x.size(0), -1)
#         x = self.fc1(x)

#         return F.log_softmax(x, dim=1)
```

In [8]:

```
def plot_data(data, label, text):
    fig = plt.figure()
    for i in range(6):
        plt.subplot(2,3,i+1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap='gray', interpolation='none')
        plt.title(text + ": {}".format(label[i]))
        plt.xticks([])
        plt.yticks([])
    plt.show()
```

In [9]:

```
def predict_batch(model, device, test_loader):
    examples = enumerate(test_loader)
    model.eval()
    with torch.no_grad():
        batch_idx, (data, target) = next(examples)
        data, target = data.to(device), target.to(device)
        output = model(data)
        pred = output.cpu().data.max(1, keepdim=True)[1] # get the index of the max log-probability
        pred = pred.numpy()
    return data.cpu().data.numpy(), target.cpu().data.numpy(), pred
```

In [10]:

```
def plot_graph(train_x, train_y, test_x, test_y, ylabel=''):
    fig = plt.figure()
    plt.plot(train_x, train_y, color='blue')
    plt.plot(test_x, test_y, color='red')
    plt.legend(['Train', 'Test'], loc='upper right')
    plt.xlabel('number of training examples seen')
    plt.ylabel(ylabel)
    plt.grid()
    plt.show()
```

In [11]:

```
def train(model, device, train_loader, optimizer, epoch, losses=[], counter=[], errors=[]):
    model.train()
    correct=0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            losses.append(loss.item())
            counter.append((batch_idx*batch_size) + ((epoch-1)*len(train_loader.dataset)))
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(target.view_as(pred)).sum().item()
    errors.append(100. * (1 - correct / len(train_loader.dataset)))
```

In [12]:

```
def test(model, device, test_loader, losses=[], errors=[]):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    losses.append(test_loss)
    errors.append(100. *  (1 - correct / len(test_loader.dataset)))
```

c) Change dataset to fashion MNIST (https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/, hint: take a look at torchvision.datasets), estimate the dataset mean and standard deviation and use it to normalize the data in the data loader.

In [13]:

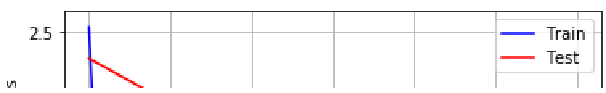
```
def main():
    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    # data transformation
    train_data = datasets.FashionMNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ]))
    test_data = datasets.FashionMNIST('../data', train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ]))

    # data loaders
    kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True, **k
wargs)
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False, **kw
args)

    # extract and plot random samples of data
    #examples = enumerate(test_loader)
    #batch_idx, (data, target) = next(examples)
    #plot_data(data, target, 'Ground truth')

    # model creation
    model = CNN3().to(device)
    # optimizer creation
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=momentum, weight_decay=wei
ght_decay)

    # lists for saving history
    train_losses = []
    train_counter = []
    test_losses = []
    test_counter = [i*len(train_loader.dataset) for i in range(epochs + 1)]
    train_errors = []
    test_errors = []
    error_counter = [i*len(train_loader.dataset) for i in range(epochs)]

    # test of randomly initialized model
    test(model, device, test_loader, losses=test_losses)

    # global training and testing loop
    for epoch in range(1, epochs + 1):
```

```
    for epoch in range(1, epochs + 1):
        train(model, device, train_loader, optimizer, epoch, losses=train_losses, counter=train_cou
nter, errors=train_errors)
        test(model, device, test_loader, losses=test_losses, errors=test_errors)

    # plotting training history
    plot_graph(train_counter, train_losses, test_counter, test_losses, ylabel='negative log likelih
ood loss')
    plot_graph(error_counter, train_errors, error_counter, test_errors, ylabel='error (%)')

    # extract and plot random samples of data with predicted labels
    data, _, pred = predict_batch(model, device, test_loader)
    plot_data(data, pred, 'Predicted')
```

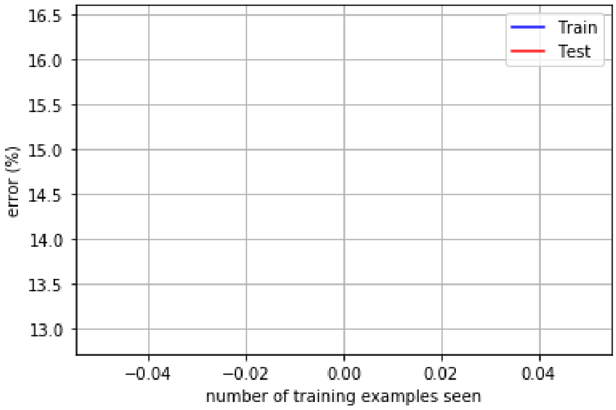
In [14]:

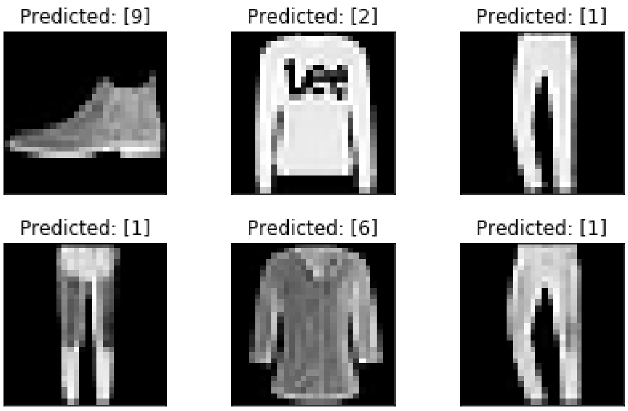
```
if __name__ == '__main__':
    main()
```

```
Test set: Average loss: 2.3017, Accuracy: 1096/10000 (11%)

Train Epoch: 1 [0/60000 (0%)] Loss: 2.529262
Train Epoch: 1 [1280/60000 (2%)] Loss: 1.037864
Train Epoch: 1 [2560/60000 (4%)] Loss: 0.832959
Train Epoch: 1 [3840/60000 (6%)] Loss: 0.771120
Train Epoch: 1 [5120/60000 (9%)] Loss: 0.489445
Train Epoch: 1 [6400/60000 (11%)] Loss: 0.910931
Train Epoch: 1 [7680/60000 (13%)] Loss: 0.640118
Train Epoch: 1 [8960/60000 (15%)] Loss: 0.752769
Train Epoch: 1 [10240/60000 (17%)] Loss: 0.552275
Train Epoch: 1 [11520/60000 (19%)] Loss: 0.514082
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.389027
Train Epoch: 1 [14080/60000 (23%)] Loss: 0.397686
Train Epoch: 1 [15360/60000 (26%)] Loss: 0.452351
Train Epoch: 1 [16640/60000 (28%)] Loss: 0.450331
Train Epoch: 1 [17920/60000 (30%)] Loss: 0.621197
Train Epoch: 1 [19200/60000 (32%)] Loss: 0.822467
Train Epoch: 1 [20480/60000 (34%)] Loss: 0.484128
Train Epoch: 1 [21760/60000 (36%)] Loss: 0.292844
Train Epoch: 1 [23040/60000 (38%)] Loss: 0.791477
Train Epoch: 1 [24320/60000 (41%)] Loss: 0.230707
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.342204
Train Epoch: 1 [26880/60000 (45%)] Loss: 0.450056
Train Epoch: 1 [28160/60000 (47%)] Loss: 0.231710
Train Epoch: 1 [29440/60000 (49%)] Loss: 0.386110
Train Epoch: 1 [30720/60000 (51%)] Loss: 0.475679
Train Epoch: 1 [32000/60000 (53%)] Loss: 0.515125
Train Epoch: 1 [33280/60000 (55%)] Loss: 0.392526
Train Epoch: 1 [34560/60000 (58%)] Loss: 0.277851
Train Epoch: 1 [35840/60000 (60%)] Loss: 0.564646
Train Epoch: 1 [37120/60000 (62%)] Loss: 0.348402
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.421125
Train Epoch: 1 [39680/60000 (66%)] Loss: 0.404908
Train Epoch: 1 [40960/60000 (68%)] Loss: 0.347592
Train Epoch: 1 [42240/60000 (70%)] Loss: 0.309540
Train Epoch: 1 [43520/60000 (72%)] Loss: 0.513286
Train Epoch: 1 [44800/60000 (75%)] Loss: 0.365399
Train Epoch: 1 [46080/60000 (77%)] Loss: 0.453077
Train Epoch: 1 [47360/60000 (79%)] Loss: 0.371936
Train Epoch: 1 [48640/60000 (81%)] Loss: 0.386824
Train Epoch: 1 [49920/60000 (83%)] Loss: 0.544805
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.236922
Train Epoch: 1 [52480/60000 (87%)] Loss: 0.410097
Train Epoch: 1 [53760/60000 (90%)] Loss: 0.364094
Train Epoch: 1 [55040/60000 (92%)] Loss: 0.703303
Train Epoch: 1 [56320/60000 (94%)] Loss: 0.267711
Train Epoch: 1 [57600/60000 (96%)] Loss: 0.319506
Train Epoch: 1 [58880/60000 (98%)] Loss: 0.235154

Test set: Average loss: 0.3614, Accuracy: 8711/10000 (87%)
```




negative log likelihood los
2.0
1.5
1.0
0.5
0
10000
20000
30000
40000
50000
60000
number of training examples seen


Train
Test
16.5
16.0
15.5
15.0
14.5
14.0
13.5
13.0
error (%)
−0.04
−0.02
0.00
0.02
0.04
number of training examples seen


Predicted: [9]
Predicted: [2]
Predicted: [1]
Predicted: [1]
Predicted: [6]
Predicted: [1]