In [3]:

```
"""
Created on Thu Mar 29 15:26:43 2018

@author: Raj Mehta
"""

from itertools import count
import torchvision
import torch
import torch.autograd
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torch.utils.data as D
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms


class data_loaderD(D.Dataset):
    def __init__(self, data, label):
        self.data = data
        self.label = label

    def __getitem__(self, index):

        return self.data[index], self.label[index]
    def __len__(self):
        return len(self.data)


def get_zeros_ones():
    train_data =datasets.MNIST('./Data/mnist', train = True, download=True,
                               transform = transforms.Compose([transforms.To
                                                               transforms.No

    train_loader = D.DataLoader(train_data, batch_size = 10000, shuffle = Tr

    X_0 = []
    X_1 = []
    X_2 = []
    y_0 = []
    y_1 = []
    y_2 = []
    for (X,y) in train_loader:
        for i in range(len(y)):
            if y[i]==0:
                X_0.append(X[i,:,:,:].numpy())
                y_0.append(y[i])
            elif y[i]==1:
                X_1.append(X[i,:,:,:].numpy())
                y_1.append(y[i])
            elif y[i]==2:
                X_2.append(X[i,:,:,:].numpy())
                y_2.append(y[i])
    X_0 = torch.FloatTensor(np.asarray(X_0)[:2000])
    X_1 = torch.FloatTensor(np.asarray(X_1)[:2000])
    X_2 = torch.FloatTensor(np.asarray(X_2)[:2000])
    y_0 = torch.FloatTensor(np.asarray(y_0)[:2000])
    y_1 = torch.FloatTensor(np.asarray(y_1)[:2000])
    y_2 = torch.FloatTensor(np.asarray(y_2)[:2000])

    X_train = torch.cat((X_0,X_1,X_2),0)
    y_train = torch.cat((y_0,y_1,y_2),0)
    print(X_train.shape)
    print(y_train.shape)
    train = data_loaderD(X_train, y_train)
    return train

#Getting Generator Input

#--------------------------------------------------------------------------



#--------------------------------------------------------------------------


class Discriminator(torch.nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 32, 5, 1, 0, bias = False) #24
        self.conv2 = torch.nn.Conv2d(32, 64, 4, 2, 1, bias = False) #12
        self.conv3 = torch.nn.Conv2d(64, 128, 5, 1, 0, bias = False) #8
        self.conv4 = torch.nn.Conv2d(128, 256, 4, 2, 1, bias = False) #4
        self.conv5 = torch.nn.Conv2d(256,  1, 4, 1, 0, bias = False) #1
        self.batchnorm1 = torch.nn.BatchNorm2d(32)
        self.batchnorm2 = torch.nn.BatchNorm2d(64)
        self.batchnorm3 = torch.nn.BatchNorm2d(128)
        self.batchnorm4 = torch.nn.BatchNorm2d(256)


    def forward(self,x):
        x = F.elu(self.batchnorm1(self.conv1(x)),0.2) #12
        x = F.elu(self.batchnorm2(self.conv2(x)),0.2) #8
        x = F.elu(self.batchnorm3(self.conv3(x)),0.2) #3
        x = F.elu(self.batchnorm4(self.conv4(x)),0.2) #1
        x = F.elu(self.conv5(x))#1
        return F.sigmoid(x)


#---------------------------------------------------------------------------


class Generator(torch.nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.deconv1 = torch.nn.ConvTranspose2d(100,256,4,1,0, bias =  False
        self.deconv2 = torch.nn.ConvTranspose2d(256,512,4,2,1, bias =  False
        self.deconv3 = torch.nn.ConvTranspose2d(512,1024,5,1,0, bias =  Fals
        self.deconv4 = torch.nn.ConvTranspose2d(1024,2048,4,2,1, bias =  Fal
        self.deconv5 = torch.nn.ConvTranspose2d(2048,1,5,1,0, bias =  False)
        self.batchnorm1 = torch.nn.BatchNorm2d(256)
        self.batchnorm2 = torch.nn.BatchNorm2d(512)
        self.batchnorm3 = torch.nn.BatchNorm2d(1024)
        self.batchnorm4 = torch.nn.BatchNorm2d(2048)


    def forward(self,x):
        x = F.elu(self.batchnorm1(self.deconv1(x))) #12
        x = F.elu(self.batchnorm2(self.deconv2(x))) #8
        x = F.elu(self.batchnorm3(self.deconv3(x))) #3
        x = F.elu(self.batchnorm4(self.deconv4(x))) #1
        x = self.deconv5(x)#1
        return F.tanh(x)



#---------------------------------------------------------------------------

#---------------------------------------------------------------------------


class Classifier(torch.nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 32, 5, 1, 0, bias = False) #24
        self.conv2 = torch.nn.Conv2d(32, 64, 4, 2, 1, bias = False) #12
        self.conv3 = torch.nn.Conv2d(64, 128, 5, 1, 0, bias = False) #8
        self.conv4 = torch.nn.Conv2d(128, 256, 4, 2, 1, bias = False) #4
        self.conv5 = torch.nn.Conv2d(256,  10, 4, 1, 0, bias = False) #1
        self.batchnorm1 = torch.nn.BatchNorm2d(32)
        self.batchnorm2 = torch.nn.BatchNorm2d(64)
        self.batchnorm3 = torch.nn.BatchNorm2d(128)
        self.batchnorm4 = torch.nn.BatchNorm2d(256)
        self.Linear1 = torch.nn.Linear(10,100)
        self.Linear2 = torch.nn.Linear(100,3)



    def forward(self,x):
        x = F.elu(self.batchnorm1(self.conv1(x)),0.2) #12
        x = F.elu(self.batchnorm2(self.conv2(x)),0.2) #8
        x = F.elu(self.batchnorm3(self.conv3(x)),0.2) #3
        x = F.elu(self.batchnorm4(self.conv4(x)),0.2) #1
        x = F.elu(self.conv5(x))#1
        x = x.view(-1,10)
        x = self.Linear1(x)
        x = self.Linear2(x)

        return F.softmax(x)


#---------------------------------------------------------------------------
gen1 = Generator()
gen2 = Generator()
gen3 = Generator()
dis = Discriminator()
```

```
cla = Classifier()


if torch.cuda.is_available():
    gen1 = gen1.cuda()
    gen2 = gen2.cuda()
    gen3 = gen3.cuda()
    dis = dis.cuda()
    cla = cla.cuda()


optimizerD = optim.SGD(dis.parameters(), lr = 0.005, momentum=0.1)
optimizerG1 = optim.SGD(gen1.parameters(), lr = 0.005, momentum=0.1)
optimizerG2 = optim.SGD(gen2.parameters(), lr = 0.005, momentum=0.1)
optimizerG3 = optim.SGD(gen3.parameters(), lr = 0.005, momentum=0.1)
optimizerC = optim.SGD(cla.parameters(), lr = 0.005, momentum = 0.1)



#---------------------------------------------------------------------------

def get_generator_data():
    noise1 = torch.FloatTensor(10, 100, 1, 1)
    noise1 = noise1.cuda()
    noise2 = torch.FloatTensor(12,100,1,1)
    noise2 = noise2.cuda()
    noise1.copy_(torch.FloatTensor(10, 100, 1, 1).normal_(0,1))
    noise2.copy_(torch.FloatTensor(12, 100, 1, 1).normal_(0,1))
    noise1v = Variable(noise1)
    noise2v = Variable(noise2)
    fake1 = gen1(noise1v)
    fake2 = gen2(noise1v)
    fake3 = gen3(noise2v)
    ygen1 = torch.zeros(10)
    ygen2 = torch.ones(10)
    ygen3 = torch.ones(12)*2
    a = 10
    b = 12
    genlabel1 = torch.cat((ygen1,ygen2,ygen3),0)
    gendata1 = torch.cat((fake1,fake2,fake3),0)

    '''
    Gdata = data_loaderD(gendata.data, genlabel)
    genloader  = D.DataLoader(Gdata, batch_size = 32, shuffle = True)
    G_k, labelg = next(iter(genloader))
    '''
    return gendata1, genlabel1.cuda()


#--------------------------------Train Classifier--------------------------

def train_classifier(epoch):
    batch_size = 32
    number = torch.FloatTensor(batch_size, 1, 28, 28)
    label = torch.LongTensor(batch_size)
    criterion = torch.nn.CrossEntropyLoss()
    if torch.cuda.is_available():
        number = number.cuda()
        label = label.cuda()
        criterion = criterion.cuda()

    train_D = get_zeros_ones()
    loader = D.DataLoader(train_D, batch_size = batch_size,drop_last = True,
    for i in range(epoch):
        for batch_idx,(X,Y) in enumerate(loader):
            cla.zero_grad()
            number.copy_(X)
            label.copy_(Y)
            numberv = Variable(number)
            label_v = Variable(label)
            output = cla(numberv)
            Loss = criterion(output, label_v)
            Loss.backward()
            print(Loss.data)
            optimizerC.step()
    print('=================Classifier Trained===============')



#--------------------------------------------------------------------------

def train(epoch):
    batch_size = 32
    train_data = get_zeros_ones()
    noise = torch.FloatTensor(batch_size, 100, 1, 1)
    fixed_noise = torch.FloatTensor(batch_size, 100, 1, 1).normal_(0,1)
    label_fake = torch.FloatTensor(batch_size)
    label_one = torch.ones(batch_size)
    label_two = torch.ones(batch_size)*2
    label_three = torch.ones(batch_size)*3
    label_zero = torch.zeros(batch_size)
    label_one = label_one.float()
    label_two = label_two.float()
    label_zero = label_zero.float()
    label_zeroL = label_zero.long()
    label_oneL = label_one.long()
    label_twoL = label_two.long()
    label_real = torch.FloatTensor(32)

    label = torch.FloatTensor(batch_size)
    image = torch.FloatTensor(batch_size, 1, 28, 28)
    criterion1 = torch.nn.CrossEntropyLoss()
    criterion2 = torch.nn.BCELoss()

    if torch.cuda.is_available():
        image = image.cuda()
        label_zero, label_one, label_two = label_zero.cuda(), label_one.cuda
        label_zeroL, label_oneL, label_twoL = label_zeroL.cuda(), label_oneL
        noise, fixed_noise = noise.cuda(), fixed_noise.cuda()
        label_real, label_fake = label_real.cuda(), label_fake.cuda()
        criterion1 = criterion1.cuda()
        criterion2 = criterion2.cuda()
```

```
train_loader = D.DataLoader(train_data, batch_size = 32, drop_last = Tru
for i in range(1, epoch+1):
    for batch_idx,(X, y) in enumerate(train_loader):
        #------------Train discriminator with real data-----------------
        dis.zero_grad()
        label_real.copy_(y.float())
        image.copy_(X)
        imagev = Variable(image)
        labelv = Variable(label_one)
        output = dis(imagev)
        real = output.data.mean()
        loss_real = criterion2(output, labelv)
        loss_real.backward()
        optimizerD.step()

        #---------------Train Discriminator with fake data of Generator
        fakev1, labelg1= get_generator_data()

        labelv = Variable(label_zero)
        output1 = dis(fakev1.detach())
        #out = clas(output0)
        loss_fake = criterion2(output1, labelv)
        #loss_clas1 = criterion1(out, Variable(label_zeroL))
        loss_fake.backward()
        #loss_clas1.backward(retain_graph = True)
        DG_z1 = output.data.mean()
        #loss = loss_real + loss_fake
        print(output.grad)
        optimizerD.step()
        print("Mean of Discriminator = %f" %DG_z1)




        '''
        #--------Train with fake data from gen2-----------
        #noise.copy_(torch.FloatTensor(32, 100, 1, 1).normal_(0,1))
        noisev = Variable(noise)
        fake2 = gen2(noisev)
        labelv = Variable(label_zero)
        output = dis(fake2.detach())
        #out = clas(output0)
        loss_fake2 = criterion2(output, labelv)
        #loss_clas2 = criterion1(out, Variable(label_oneL))
        loss_fake2.backward()
        #loss_clas2.backward(retain_graph = True)
        DG_z1 = output.data.mean()
        #loss = loss_real + loss_fake

        #--------Train with fake data from gen3-----------
        #noise.copy_(torch.FloatTensor(32, 100, 1, 1).normal_(0,1))
        noisev = Variable(noise)
        fake3 = gen3(noisev)
        labelv = Variable(label_zero)
        output = dis(fake3.detach())
        #out = clas(output0)
        loss_fake3 = criterion2(output, labelv)
        #loss_clas3 = criterion1(out, Variable(label_twoL))
        loss_fake3.backward()
        #loss_clas3.backward()
        DG_z1 = output.data.mean()
        #loss = loss_real + loss_fake
        optimizerD.step()
        #optimizerC.step()

        '''

        #------------------------------------Generator with discrim
        gen1.zero_grad()
        gen2.zero_grad()
        gen3.zero_grad()
        labelv = Variable(label_one)
        output1 = dis(fakev1)
        loss_G1 = criterion2(output1[:10], labelv[:10])

        #loss_Gclas1 = criterion1(out, Variable(label_zeroL))
        loss_G1.backward(retain_graph = True)
        optimizerG1.step()
        #loss_Gclas1.backward()
        loss_G2 = criterion2(output1[10:20], labelv[10:20])

        loss_G2.backward(retain_graph = True)
        optimizerG2.step()

        loss_G3 = criterion2(output1[20:], labelv[20:])

        loss_G3.backward(retain_graph = True)
        optimizerG3.step()


        DG1_z2 = output1.data.mean()

        print("Mean of generators = %f" %DG1_z2)


        #------------------------------------Generator with class
        label_v = Variable(label_zeroL)
        output2 = cla(fakev1)
        loss_cG1 = criterion1(output2[:10], label_v[:10])

        #loss_Gclas1 = criterion1(out, Variable(label_zeroL))
        loss_cG1.backward(retain_graph = True)
        optimizerG1.step()
        #loss_Gclas1.backward()
        label_v = V ariable(label_oneL)
        loss_cG2 = criterion1(output2[10:20], label_v[10:20])

        loss_cG2.backward(retain_graph = True)
        optimizerG2.step()

        label_v = Variable(label_twoL)
        loss_cG3 = criterion1(output2[20:], label_v[20:])

        loss_cG3.backward()
```

```
            optimizerG3.step()







            '''

            #------------------------------------Generator 2
            gen2.zero_grad()
            labelv = Variable(label_one)
            output = dis(fake2)
            loss_G2 = criterion2(output, labelv)
            #loss_Gclas1 = criterion1(out, Variable(label_zeroL))
            loss_G2.backward()
            #loss_Gclas1.backward()
            DG1_z2 = output.data.mean()



            #------------------------------------Generator 3
            gen3.zero_grad()
            labelv = Variable(label_one)
            output = dis(fake3)
            loss_G3 = criterion2(output, labelv)
            #loss_Gclas1 = criterion1(out, Variable(label_zeroL))
            loss_G3.backward()
            #loss_Gclas1.backward()
            DG1_z2 = output.data.mean()
            optimizerG1.step()
            optimizerG2.step()
            optimizerG3.step()
            '''
            if(batch_idx%10):
              print('batch %d'%batch_idx)
        print('---------end of epoch %d -----------' %i)

train_classifier(6)

train(5)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
 (http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (h
ttp://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz (h
ttp://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz)
Processing...
Done!
torch.Size([6000, 1, 28, 28])
torch.Size([6000])
```

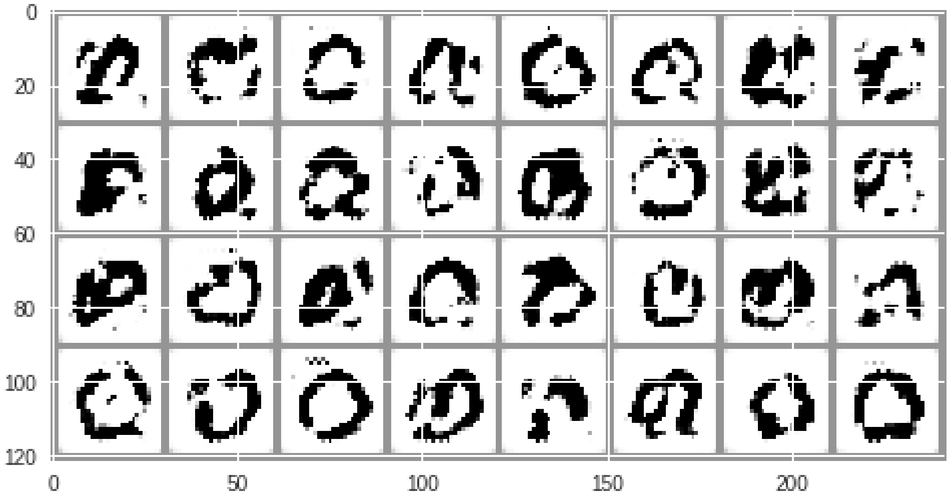
```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:155: UserWar
ning: Implicit dimension choice for softmax has been deprecated. Change t
he call to include dim=X as an argument.
```

In [4]:

```
fake = gen1(Variable(torch.FloatTensor(32, 100, 1, 1).normal_(0,1).cuda()))
a = torchvision.utils.make_grid(fake.data)
plt.imshow(a[0,:,:])
```

Out[4]:

```
<matplotlib.image.AxesImage at 0x7f84ac8515c0>
```



In [0]:

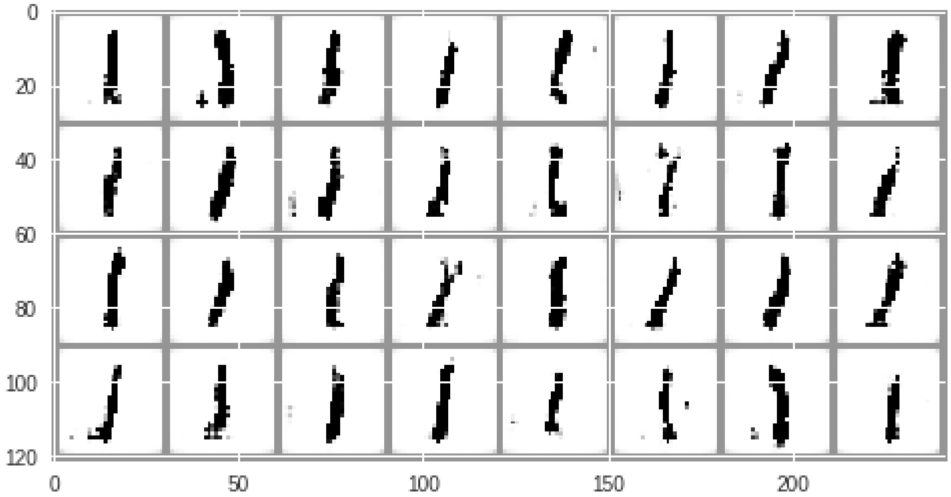
```
3# http://pytorch.org/
from os import path
from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())

accelerator = 'cu80' if path.exists('/opt/bin/nvidia-smi') else 'cpu'

!pip install -q http://download.pytorch.org/whl/{accelerator}
    /torch-0.3.0.post4-{platform}-linux_x86_64.whl torchvision
import torch
```

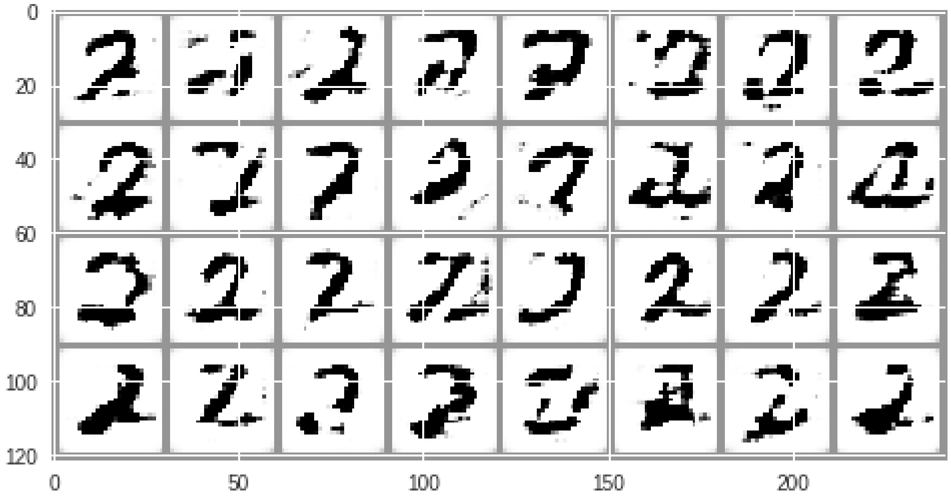
```
In [5]: fake = gen2(Variable(torch.FloatTensor(32, 100, 1, 1).normal_(0,1).cuda()))
        a = torchvision.utils.make_grid(fake.data)
        plt.imshow(a[0,:,:])
```

```
Out[5]: <matplotlib.image.AxesImage at 0x7f84ac7c24e0>
```



```
In [7]: fake = gen3(Variable(torch.FloatTensor(32, 100, 1, 1).normal_(0,1).cuda()))
        a = torchvision.utils.make_grid(fake.data)
        plt.imshow(a[0,:,:])
```

```
Out[7]: <matplotlib.image.AxesImage at 0x7f84acb8e668>
```



```
In [0]:
```