IFT6135 – Representation Learning

Winter 2019

Assignment 1 – Practical Part:

Multilayer Perceptrons and Convolutional Neural Networks

By:

Name	Matricule
Jonathan Plante	20016047
Laura Gagliano	20056833
Martin Weiss	
Bernard Dadjio	

Presented to: Prof. Aaron Courville

Git: https://github.com/jonPlante/IFT6135_Ass1

Kaggle : Jonathan

February 17 2019

Problem 1

February 17, 2019

google collab link: https://colab.research.google.com/drive/155TdCXJxHunKuByI8kIaX_P-XQic0qLd

You must download the minst data set from git and upload it to your google drive:
https://github.com/jonPlante/IFT6135_Ass1

```
In [ ]: import numpy as np
        import gzip
        import pickle
        import matplotlib.pyplot as plt
        from google.colab import drive
        drive.mount('/content/gdrive')
```

```
In [0]: # load data
        with gzip.open('/content/gdrive/My Drive/mnist.pkl.gz', 'rb') as f:
            train_set, valid_set, test_set = pickle.load(f, encoding='latin1')
```

0.1 Building the model

```
In [0]: class NN(object):

            def __init__(self,Winitial,train_set,valid_set,test_set,LR,hidden_dims,n_hidden=2,r
              self.train_set=train_set
              self.valid_set=valid_set
              self.test_set=test_set
              self.LR=LR
              self.dims=hidden_dims
              self.initialize_weights(Winitial,hidden_dims,len(self.train_set[0][0]),np.max(se
              self.reset_grad(len(self.train_set[0][0]))


            def initialize_weights(self,Winitial,dims,inputs,n_classes):
              if Winitial=='zero':
                self.W1=np.zeros((dims[0],inputs))
                self.W2=np.zeros((dims[1],dims[0]))
                self.W3=np.zeros((n_classes,dims[1]))

              elif Winitial=='normal':
                self.W1=np.random.normal(0,1,(dims[0],inputs))
```

```
    self.W2=np.random.normal(0,1,(dims[1],dims[0]))
    self.W3=np.random.normal(0,1,(n_classes,dims[1]))

  elif Winitial=='glarot':
    dl=np.sqrt(6/(dims[0]+inputs))
    self.W1=np.random.uniform(-dl,dl,(dims[0],inputs))
    dl=np.sqrt(6/(dims[1]+dims[0]))
    self.W2=np.random.uniform(-dl,dl,(dims[1],dims[0]))
    dl=np.sqrt(6/(n_classes+dims[1]))
    self.W3=np.random.uniform(-dl,dl,(n_classes,dims[1]))


  self.b1=np.zeros((dims[0],1))
  self.b2=np.zeros((dims[1],1))
  self.b3=np.zeros((n_classes,1))
  self.n_classes=n_classes
def forward(self,input,label):
  self.oneHot=np.zeros((self.n_classes,1))
  self.oneHot[label]=1
  self.a1=np.dot(self.W1,input)+self.b1
  self.h1=self.activation_sig(self.a1)
  self.a2=np.dot(self.W2,self.h1)+self.b2
  self.h2=self.activation_sig(self.a2)
  self.a3=np.dot(self.W3,self.h2)+self.b3
  self.output=self.softmax(self.a3)
  self.L=self.loss(self.output,self.oneHot)


def activation(self,input):
  #relu
  return np.maximum(0,input)
def activation_sig(self,input):
  return 1/(1+np.exp(-input))

def loss(self,predictions,onehot):
  return np.sum(-onehot*np.log(predictions))

def softmax(self,input):
  input=input-np.max(input)
  return np.exp(input)/np.sum(np.exp(input))

def reset_grad(self,inputs):
  self.gW3_MB=np.zeros((self.n_classes,self.dims[1]))
  self.gb3_MB=np.zeros((self.n_classes,1))
  self.gW2_MB=np.zeros((self.dims[1],self.dims[0]))
  self.gb2_MB=np.zeros((self.dims[1],1))
  self.gW1_MB=np.zeros((self.dims[0],inputs))
  self.gb1_MB=np.zeros((self.dims[0],1))
```

```
def backward(self,input):
  self.go=-1/self.output*self.oneHot
  self.ga3=self.go[np.argmax(self.oneHot)]*(self.oneHot*self.output-self.output[np
  self.gW3=self.ga3*self.h2.reshape(1,-1)
  self.gb3=self.ga3
  self.gh2=np.dot(np.transpose(self.W3),self.ga3)
  self.ga2=self.gh2*self.h2*(1-self.h2)
  self.gW2=self.ga2*self.h1.reshape(1,-1)
  self.gb2=self.ga2
  self.gh1=np.dot(np.transpose(self.W2),self.ga2)
  self.ga1=self.gh1*self.h1*(1-self.h1)
  self.gW1=self.ga1*input.reshape(1,-1)
  self.gb1=self.ga1

  self.gW3_MB+=self.gW3/self.MB_size
  self.gb3_MB+=self.gb3/self.MB_size
  self.gW2_MB+=self.gW2/self.MB_size
  self.gb2_MB+=self.gb2/self.MB_size
  self.gW1_MB+=self.gW1/self.MB_size
  self.gb1_MB+=self.gb1/self.MB_size

def update(self):
  self.W1=self.W1-self.LR*self.gW1_MB
  self.b1=self.b1-self.LR*self.gb1_MB
  self.W2=self.W2-self.LR*self.gW2_MB
  self.b2=self.b2-self.LR*self.gb2_MB
  self.W3=self.W3-self.LR*self.gW3_MB
  self.b3=self.b3-self.LR*self.gb3_MB

def train(self,epoch,MB):
  self.train_data=np.zeros((epoch+1,3))
  self.valid_data=np.zeros((epoch+1,3))
  self.reset_grad(len(self.train_set[0][0]))
  self.MB_size=MB
  Tot_Loss=0
  example_count=0
  print('Computing initial training error and loss')
  train_Loss, train_error=self.test(self.train_set[0],self.train_set[1])
  print('Initial training loss: ' +str(np.round(train_Loss,2))+', initial train er
  print('Computing initial validation error and loss')
  valid_Loss, valid_error=self.test(self.valid_set[0],self.valid_set[1])
  print('Initial validation loss: ' +str(np.round(valid_Loss,2))+', initial valid
  self.train_data[0,:]=[0,train_Loss,train_error]
  self.valid_data[0,:]=[0,valid_Loss,valid_error]
  for k in range(epoch):
    for j in range(len(self.train_set[0])):
      self.forward(np.array(self.train_set[0][j]).reshape(-1,1),self.train_set[1][
```

```
        self.backward(np.array(self.train_set[0][j]).reshape(-1,1))
        Tot_Loss+=self.L
        example_count+=1
        if j%MB==0 and not j==0:
          self.update()
          self.reset_grad(len(self.train_set[0][0]))

        if j%1000==0 and not j==0:
          print('\tLoss: '+str(np.round(Tot_Loss/example_count,3))+' '+str(np.round(j
          Tot_Loss=0
          example_count=0
      print('Epoch: '+str(k+1) +' complete\n')
      print('Computing training error and loss')
      train_Loss, train_error=self.test(self.train_set[0],self.train_set[1])
      print('Training loss: ' +str(np.round(train_Loss,2))+', train error: '+str(np.r
      print('Computing validation error and loss')
      valid_Loss, valid_error=self.test(self.valid_set[0],self.valid_set[1])
      print('Validation loss: ' +str(np.round(valid_Loss,2))+', valid error: '+str(np
      self.train_data[k+1,:]=[k+1,train_Loss,train_error]
      self.valid_data[k+1,:]=[k+1,valid_Loss,valid_error]

    test_Loss, test_error=self.test(self.test_set[0],self.test_set[1])
    print('\nTraining complete\nError on test set: '+str(np.round(test_error*100,2))+

  def test(self,input,label):
    sumLoss=0
    incorrect=0
    for k in range(len(input)):
      self.forward(np.array(input[k]).reshape(-1,1),label[k])
      sumLoss+=self.L
      incorrect+=np.minimum(1,np.abs(label[k]-np.argmax(self.output)))
      if k%1000==0:
          print('\t'+str(np.round(k/len(input)*100,2))+'% of calc complete')
    return sumLoss/len(input), incorrect/len(input)

  def predict(self,input):
    a1=np.dot(self.W1,input)+self.b1
    h1=self.activation_sig(a1)
    a2=np.dot(self.W2,h1)+self.b2
    h2=self.activation_sig(a2)
    a3=np.dot(self.W3,h2)+self.b3
    output=self.softmax(a3)
    return np.argmax(output)
```

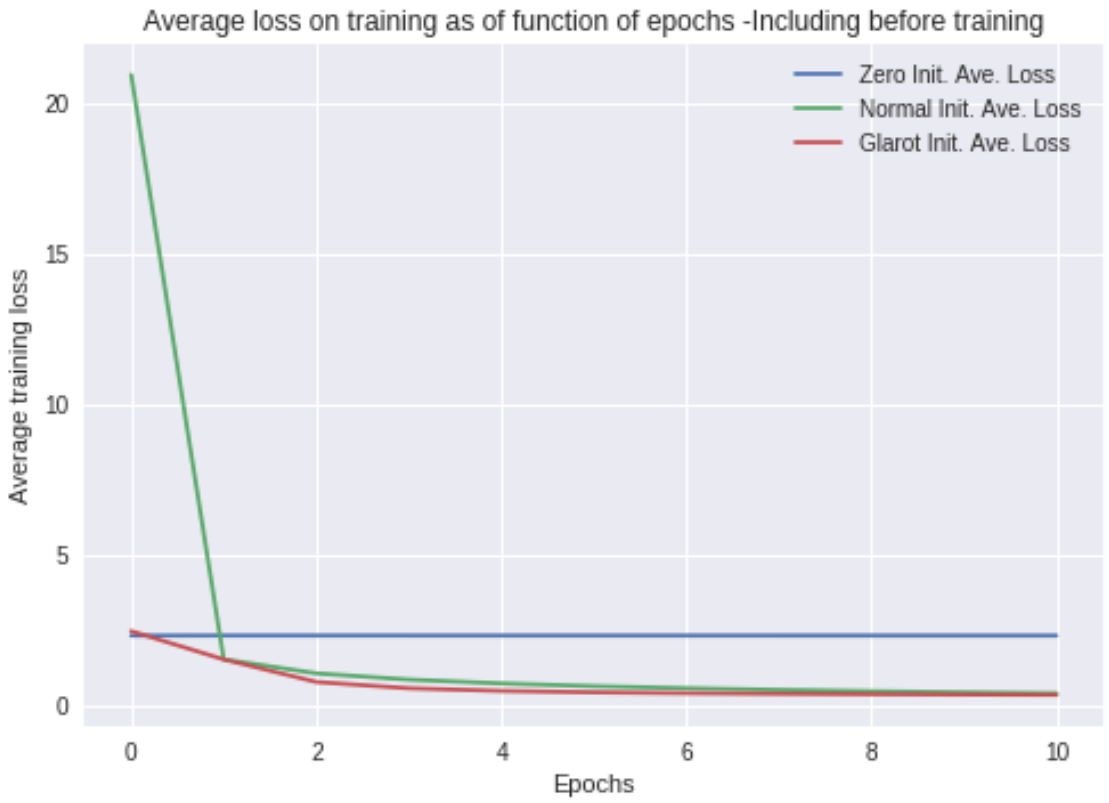
0.2 Initialization

Number of hidden units in layer 1 is 520. Number of hidden units in layer 2 is 480.
The total set of parameters is 784x520+520+520x480+480+480x10+10=663 090 = 0.66309 M.
The activation is the sigmoid function. The learning rate is 0.05 and the minibatch size is 100.

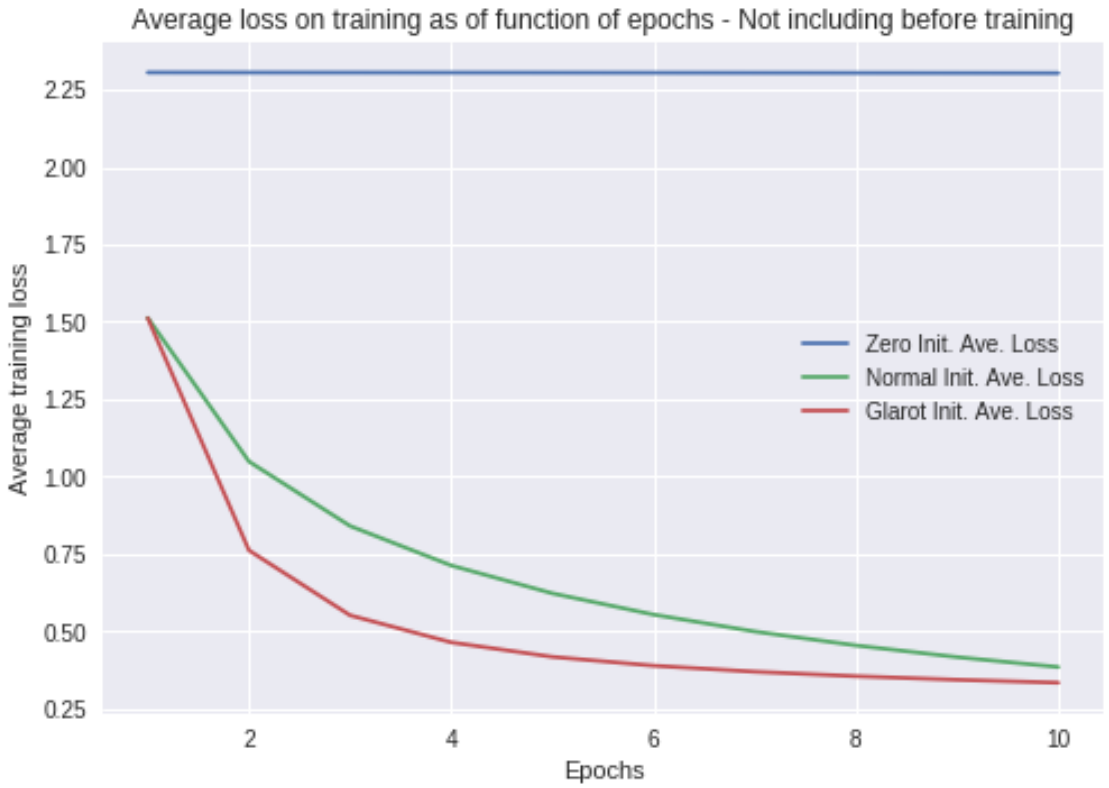
```
In [ ]: LR=0.05
        dims=(520,480)
        MB=100
        epochs=10
        print('Zero initialization\n')

        NN_zero=NN('zero',train_set,valid_set,test_set,LR,dims)
        NN_zero.train(epochs,MB)

        print('Normal initialization\n')
        NN_normal=NN('normal',train_set,valid_set,test_set,LR,dims)
        NN_normal.train(epochs,MB)

        print('Glarot initialization\n')
        NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
        NN_glarot.train(epochs,MB)


        print('\nPlotting results\n')
        plt.plot(NN_zero.train_data[:,0],NN_zero.train_data[:,1],label='Zero Init. Ave. Loss')
        plt.plot(NN_normal.train_data[:,0],NN_normal.train_data[:,1],label='Normal Init. Ave. L
        plt.plot(NN_glarot.train_data[:,0],NN_glarot.train_data[:,1],label='Glarot Init. Ave. L
        plt.title('Average loss on training as of function of epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Average training loss')
        plt.legend()
        plt.show()



In [0]: print('\nPlotting results\n')
        plt.plot(NN_zero.train_data[:,0],NN_zero.train_data[:,1],label='Zero Init. Ave. Loss')
        plt.plot(NN_normal.train_data[:,0],NN_normal.train_data[:,1],label='Normal Init. Ave. L
        plt.plot(NN_glarot.train_data[:,0],NN_glarot.train_data[:,1],label='Glarot Init. Ave. L
        plt.title('Average loss on training as of function of epochs -Including before training
        plt.xlabel('Epochs')
        plt.ylabel('Average training loss')
        plt.legend()
        plt.show()

        print('\nPlotting results\n')
        plt.plot(NN_zero.train_data[1:,0],NN_zero.train_data[1:,1],label='Zero Init. Ave. Loss
        plt.plot(NN_normal.train_data[1:,0],NN_normal.train_data[1:,1],label='Normal Init. Ave
```

```
plt.plot(NN_glarot.train_data[1:,0],NN_glarot.train_data[1:,1],label='Glarot Init. Ave
plt.title('Average loss on training as of function of epochs - Not including before tra
plt.xlabel('Epochs')
plt.ylabel('Average training loss')
plt.legend()
plt.show()
```

Plotting results



Plotting results



As one can observe from the graphs above, when initializing the weights with the "zero" method, the training loss remains constant. This is natural since backpropagation cannot get through the hidden units, since we multiply the last hidden layers gradient by the weights of zero which leads to zero gradient. The only parameters that can improve are the bias of the last layer.

We can also observe that when initializing with the "Normal" and "Glarot" methods, there is learning occuring since the average training loss diminishes. We observe that the the Glarot method diminishes training loss more rapidly and therefore is a better initialisation of the weights.

1 Hyperparameter search

In [ ]:
```
#try 1

LR=0.1
dims=(520,480)
MB=100
epochs=10

print('Glarot initialization\n')
NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
NN_glarot.train(epochs,MB)
```

In [ ]:
```
#try 2

LR=0.1
dims=(520,480)
MB=200
epochs=10

print('Glarot initialization\n')
NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
NN_glarot.train(epochs,MB)
```

In [ ]:
```
#try 3

LR=0.1
dims=(520,480)
MB=50
epochs=10

print('Glarot initialization\n')
NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
NN_glarot.train(epochs,MB)
```

In [ ]:
```
#try 4

LR=0.1
dims=(550,480)
MB=50
epochs=10

print('Glarot initialization\n')
NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
NN_glarot.train(epochs,MB)
```

In [ ]:
```
#try 5

LR=0.1
dims=(520,550)
MB=50
epochs=10

print('Glarot initialization\n')
NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
NN_glarot.train(epochs,MB)
```

In [ ]:
```
#try 6

LR=0.15
dims=(520,600)
MB=35
```

```
        epochs=10

        print('Glarot initialization\n')
        NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
        NN_glarot.train(epochs,MB)
```

```
In [ ]: LR=0.30
        dims=(520,600)
        MB=20
        epochs=10

        print('Glarot initialization\n')
        NN_glarot=NN('glarot',train_set,valid_set,test_set,LR,dims)
        NN_glarot.train(epochs,MB)
```

An average of 97.06% accuracy on the validation set and an average of 96.97 % accuracy on the test set was observed with the following parameters:

Hidden units of layer 1: 520
Hidden units of layer 2: 600
Learning rate: 0.30
Mini batch size : 20
Nonlinearity used: sigmoid
The following table shows the hyperparamater search that was done:

```
In [1]: from IPython.display import Image
        Image(filename='JPtable.png')
```

Out[1]:

Try	Hidden units in layer 1	Hidden units in layer 2	Learning rate	Mini batch size	Nonlinearity	Accuracy on validation set (%)
1	520	480	0.05	100	Sigmoid	90.93
2	520	480	0.10	100	Sigmoid	91.71
3	520	480	0.10	200	Sigmoid	91.08
4	520	480	0.10	50	Sigmoid	93.37
5	520	480	0.10	50	Sigmoid	93.30
6	520	550	0.10	50	Sigmoid	93.38
7	520	600	0.15	35	Sigmoid	95.41
8	520	600	0.30	20	Sigmoid	97.06

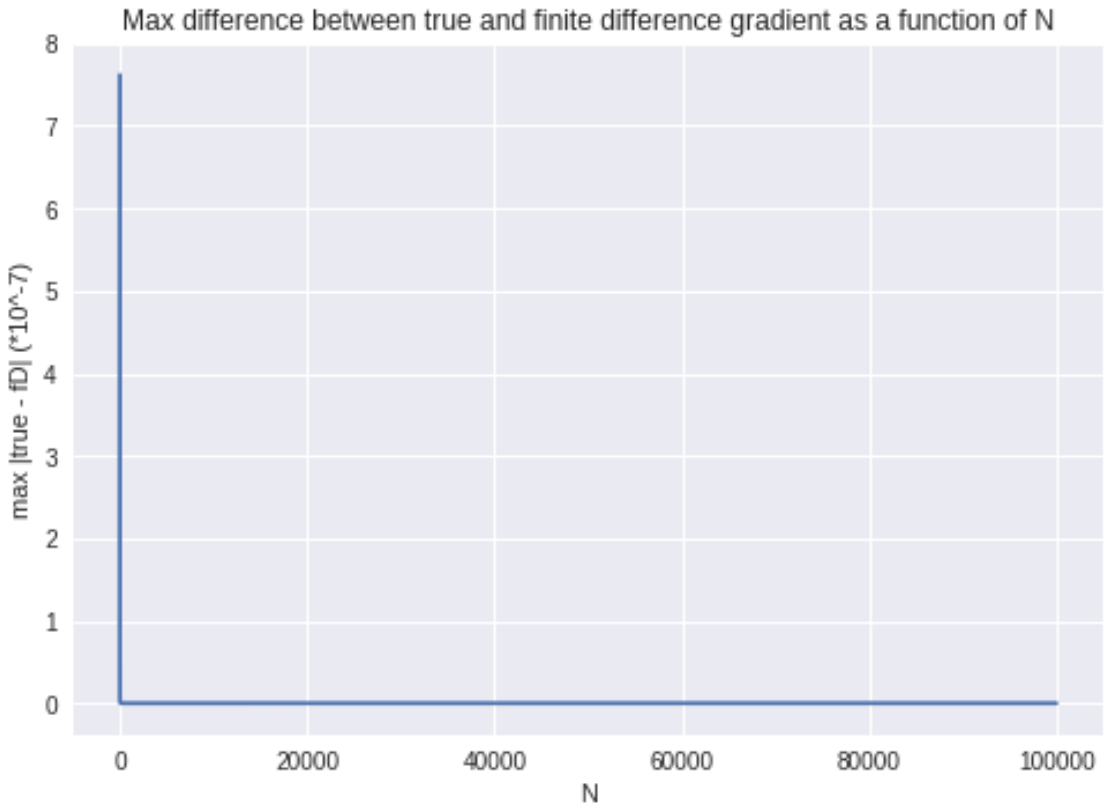
2 Validate gradients using finite different

```
In [ ]: #get gradient for one training example
        NN_glarot.forward(np.array(NN_glarot.train_set[0][0]).reshape(-1,1),NN_glarot.train_se
```

```
        NN_glarot.backward(np.array(NN_glarot.train_set[0][0]).reshape(-1,1))

        noPoints=100
        true_grad=NN_glarot.gW2[0,:10]
        finite_grad=np.zeros((noPoints,10)) # values by parameters N*I
        iValues=np.linspace(0,5,noPoints)
        N=np.zeros(noPoints)
        for i in range(noPoints):
          n=10**iValues[i]
          N[i]=n
          for j in range(10):
            weight=NN_glarot.W2[0,j]
            NN_glarot.W2[0,j]=weight+1/n
            NN_glarot.forward(np.array(NN_glarot.train_set[0][0]).reshape(-1,1),NN_glarot.trai
            loss1=NN_glarot.L
            NN_glarot.W2[0,j]=weight-1/n
            NN_glarot.forward(np.array(NN_glarot.train_set[0][0]).reshape(-1,1),NN_glarot.trai
            loss2=NN_glarot.L
            NN_glarot.W2[0,j]=weight
            finite_grad[i,j]=(loss1-loss2)/(2*1/n)

        print("done")
```

In [0]:
```
        maxDiff=np.amax(np.abs(finite_grad-true_grad),axis=1)
        maxDiff=maxDiff/(10**(-7))
        print('\nPlotting results\n')
        plt.plot(N,maxDiff)
        plt.title('Max difference between true and finite difference gradient as a function of
        plt.xlabel('N')
        plt.ylabel('max |true - fD| (*10^-7)')
        plt.show()
```

```
Plotting results
```



We can see from the figure above that when N gets bigger, the difference between the true gradient and finite difference gradient decreases. We also observe that the difference for both gradients is extremely small (10^-7). This is a sign that the Neural Net has converged. If we had done this before training the neural net, the difference would be much greater.

Problem 2

February 17, 2019

https://colab.research.google.com/drive/1eLLhRCl2VRY00EJVoeFaij5-r2QVl_0I
You must download the minst data set from git and upload it to your google drive:
https://github.com/jonPlante/IFT6135_Ass1)

```
In [ ]: import numpy as np
        import gzip
        import pickle
        import matplotlib.pyplot as plt
        from google.colab import drive
        drive.mount('/content/gdrive')

In [0]: # load data
        with gzip.open('/content/gdrive/My Drive/mnist.pkl.gz', 'rb') as f:
            train_set, valid_set, test_set = pickle.load(f, encoding='latin1')

In [ ]: #install pytorch
        !pip3 install http://download.pytorch.org/whl/cu92/torch-0.4.1-cp36-cp36m-linux_x86_64
        !pip3 install torchvision

In [0]: import torch
        import torchvision
        import torchvision.transforms as transforms
        from torch.autograd import Variable
        from torch.utils.data.dataset import Dataset

In [0]: #convert data in to pytorch format

        mnist_train=[]
        mnist_valid=[]
        mnist_test=[]
        for i in range(len(train_set[0])):
          x=torch.Tensor(train_set[0][i].reshape((1,28,28)))
          mnist_train.append([x,train_set[1][i]])

        for i in range(len(valid_set[0])):
          x=torch.Tensor(valid_set[0][i].reshape((1,28,28)))
          mnist_valid.append([x,valid_set[1][i]])
```

```
for i in range(len(test_set[0])):
  x=torch.Tensor(test_set[0][i].reshape((1,28,28)))
  mnist_test.append([x,test_set[1][i]])
```

1 CNN Architecture

The first layer is 120 kernels with size of 4x4 with 1 channel (1920 parameters). The activation is Relu followed by with a non overlapping max pool layer of 2x2.

The second layer is 300 kernels with size 4x4 with 120 channels (576, 000 parameters). The activation is Relu followed by a non overlapping max pool layer of 2x2.

We then have a fully connected layer with bias of 20 neurons with 4800 inputs (96, 020 parameters). The activation is Relu.

Finally we have a second fully connected with bias layer of 10 neurons with 20 inputs (210 parameters) leading to the outputs.

The total amount of parameters is 674, 150 = 0.67415 M.

The mini batch size is 50 and the learning rate 0.01.

```
In [0]: import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim


        class Net(nn.Module):
            def __init__(self):
                super(Net, self).__init__()
                self.conv1 = nn.Conv2d(1, 120, 4)
                self.pool = nn.MaxPool2d(2, 2)
                self.conv2 = nn.Conv2d(120, 300, 4)
                self.fc1 = nn.Linear(300 * 4 * 4, 20)
                self.fc2 = nn.Linear(20, 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, 300 * 4 * 4)
                x = F.relu(self.fc1(x))
                x = self.fc2(x)
                return x
```

```
In [ ]: epochs=10
        MB=50
        LR=0.01
        # using cuda
        use_cuda = True
        net = Net()
        if use_cuda and torch.cuda.is_available():
            net.cuda()
            print('using cuda')
```

```
else:
  print('not using cuda')

trainloader = torch.utils.data.DataLoader(mnist_train, batch_size=MB,shuffle=False, nu
validloader = torch.utils.data.DataLoader(mnist_valid, batch_size=MB,shuffle=False, nu
testloader = torch.utils.data.DataLoader(mnist_test, batch_size=MB,shuffle=False, num_

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.0)
train_data=np.zeros((epochs+1,3))
valid_data=np.zeros((epochs+1,3))

print('Calculating initial training loss')
total_loss = 0.0
total_error=0
for i, data in enumerate(trainloader, 0):
  # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)
  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()
  outputs = net(images)
  loss = criterion(outputs, labels)
  total_error+=np.sum(np.minimum(np.abs((torch.max(outputs,1)[1]-labels).data.cpu().nu
  total_loss+=loss.data.cpu()
  if i*MB%10000==0 and not i==0:
    print('\t'+str(np.round(i*MB/50000*100,2))+'% complete')
train_data[0,:]=[0,total_loss/(50000/MB),total_error/50000]
print('Initial training loss: ' +str(np.round(train_data[0,1],2))+', training error: '

print('Calculating initial validation loss')
total_loss = 0.0
total_error=0
for i, data in enumerate(validloader, 0):
  # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)
  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()
  outputs = net(images)
  loss = criterion(outputs, labels)
  total_error+=np.sum(np.minimum(np.abs((torch.max(outputs,1)[1]-labels).data.cpu().nu
  total_loss+=loss.data.cpu()
valid_data[0,:]=[0,total_loss/(10000/MB),total_error/10000]
```

```
print('Initial validation loss: ' +str(np.round(valid_data[0,1],2))+', validation erro

for epoch in range(epochs):  # loop over the dataset multiple times
  running_loss = 0.0
  num_ex=0
  for i, data in enumerate(trainloader, 0):
    # get the inputs
    images, labels = data
    images=Variable(images)
    labels=Variable(labels)
    if use_cuda and torch.cuda.is_available():
      images = images.cuda()
      labels = labels.cuda()
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = net(images)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    # print statistics
    running_loss += loss.data.cpu().numpy()
    num_ex+=1
    if i*MB%10000 == 0 and not i==0:
      print('\tLoss: '+str(np.round(running_loss/num_ex,3))+', '+str(np.round(i*MB/500
      num_ex=0
      running_loss=0
  print('Epoch '+str(epoch+1) + ' complete')

  print('Calculating training loss')
  total_loss = 0.0
  total_error=0
  for i, data in enumerate(trainloader, 0):
    # get the inputs
    images, labels = data
    images=Variable(images)
    labels=Variable(labels)
    if use_cuda and torch.cuda.is_available():
      images = images.cuda()
      labels = labels.cuda()
    outputs = net(images)
    loss = criterion(outputs, labels)
    total_error+=np.sum(np.minimum(np.abs((torch.max(outputs,1)[1]-labels).data.cpu().
    total_loss+=loss.data.cpu()
    if i*MB%10000==0 and not i==0:
      print('\t'+str(np.round(i*MB/50000*100,2))+'% complete')
  train_data[epoch+1,:]=[epoch+1,total_loss/(50000/MB),total_error/50000]
  print('Training loss: ' +str(np.round(train_data[epoch+1,1],2))+', training error: '
```

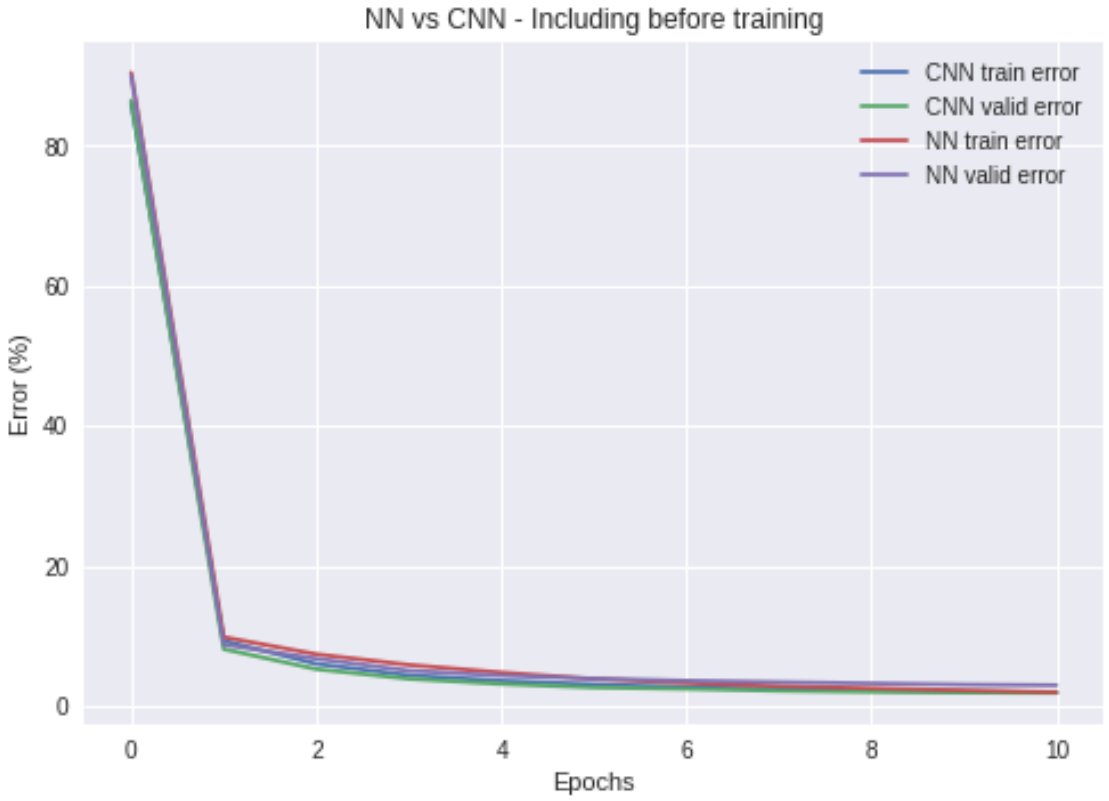
```
    print('Calculating validation loss')
    total_loss = 0.0
    total_error=0
    for i, data in enumerate(validloader, 0):
      # get the inputs
      images, labels = data
      images=Variable(images)
      labels=Variable(labels)
      if use_cuda and torch.cuda.is_available():
        images = images.cuda()
        labels = labels.cuda()
      outputs = net(images)
      loss = criterion(outputs, labels)
      total_error+=np.sum(np.minimum(np.abs((torch.max(outputs,1)[1]-labels).data.cpu().
      total_loss+=loss.data.cpu()
    valid_data[epoch+1,:]=[epoch+1,total_loss/(10000/MB),total_error/10000]
    print('Validation loss: ' +str(np.round(valid_data[epoch+1,1],2))+', validation erro

  print('Finished Training')
  torch.cuda.empty_cache()

  print('Calculating performance on test set')
  total_error=0
  for i, data in enumerate(testloader, 0):
  # get the inputs
    images, labels = data
    images=Variable(images)
    labels=Variable(labels)
    if use_cuda and torch.cuda.is_available():
      images = images.cuda()
      labels = labels.cuda()
    outputs = net(images)
    total_error+=np.sum(np.minimum(np.abs((torch.max(outputs,1)[1]-labels).data.cpu().num

  print('Test error: '+str(np.round(total_error/10000*100,2))+'%, test accuracy:'+str(np
```

2 CNN vs MLP

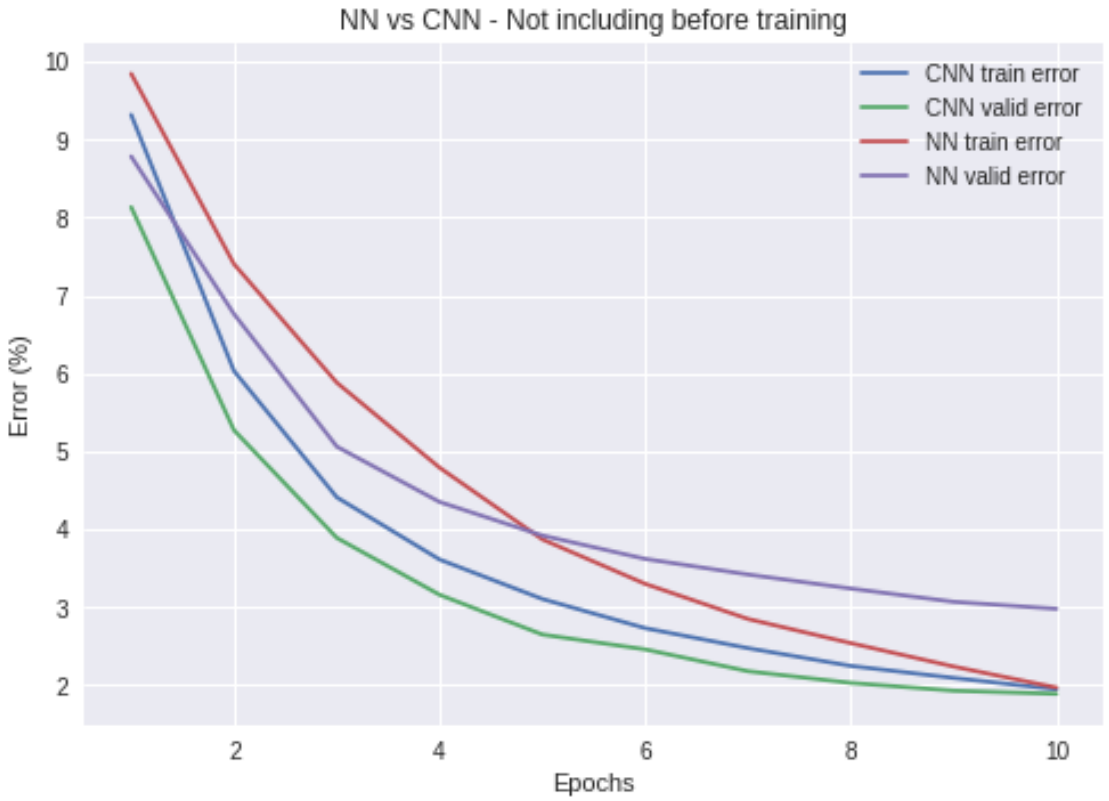
```
In [0]: NN_data_train=np.array([90.32,9.84,7.40,5.88,4.79,3.87,3.30,2.85,2.54,2.24,1.97])
        NN_data_valid=np.array([89.91,8.78,6.76,5.06,4.35,3.92,3.62,3.42,3.24,3.07,2.98])
        print('\nPlotting results\n')
        plt.plot(train_data[:,0],train_data[:,2]*100,label='CNN train error')
        plt.plot(valid_data[:,0],valid_data[:,2]*100,label='CNN valid error')
        plt.plot(train_data[:,0],NN_data_train,label='NN train error')
        plt.plot(valid_data[:,0],NN_data_valid,label='NN valid error')
```

```
plt.title('NN vs CNN - Including before training')
plt.xlabel('Epochs')
plt.ylabel('Error (%)')
plt.legend()
plt.show()

print('\nPlotting results\n')
plt.plot(train_data[1:,0],train_data[1:,2]*100,label='CNN train error')
plt.plot(valid_data[1:,0],valid_data[1:,2]*100,label='CNN valid error')
plt.plot(train_data[1:,0],NN_data_train[1:],label='NN train error')
plt.plot(valid_data[1:,0],NN_data_valid[1:],label='NN valid error')
plt.title('NN vs CNN - Not including before training')
plt.xlabel('Epochs')
plt.ylabel('Error (%)')
plt.legend()
plt.show()
```

```
Plotting results
```



```
Plotting results
```



CNN= Convolutional Neural Network, NN= Multi Layer Perceptron

From the graphs above, one can see that the CNN error rate drops faster. We can also see that the the generalisation gap between the validation error and training error for the CNN after 10 epochs is very small (less than 0.1%) while the gap between the training error and validation error for the NN is much bigger (about 1%). We also see that the NN and CNN training error after 10 epochs is 2%, while the validation error for CNN is 2% and for NN is 3%.

Finally, the test error on CNN was 2% while on NN it was about 3%. This shows that the CNN generalises better that the NN for images since the test, validation and training error are all the same. While in the NN the test and validation error increased by 1% from the training error.

Problem 3

Kaggle submission: Jonathan

1. Final architecture chosen

The model of the submitted binary classification algorithm is composed of 2 main parts: the the convolutional neural network classifier definition and the neural network training process. We will begin with the definition of the CNN architecture which was inspired by the popular ResNet network. As described in the table below, the network contains a total of 36 layers where 33 are convolutional layers, 1 is an average pooling layer, and 2 are fully connected layers. The activation function used is the rectified linear unit (ReLU) and the reason for many convolutional layers is that while designing the network, classification performance was strongly modulated by the number of layers. However, large numbers of layers often puts the network at risk for the vanishing gradient problem. To remedy this, a residual approach was used during the forward propagation. The total number of parameters is about 21.6 M. Table 3.1 describes the full architecture while figure 3.1 and 3.2 display the residual approach and the architecture.

Table 3.1: Detailed architecture of convolutional neural network classifier (dilution =1)

Layer	Input Size	Output Size	Input channels	Output channels	Mask size	stride	Pad	Parameters
Conv 1	64	64	3	64	3	1	1	576
Conv 2	64	64	64	64	3	1	1	12,288
Conv 3	64	64	64	64	3	1	1	12,288
Conv 3 -Res	-	-	-	-	-	-	-	-
Conv 4	64	64	64	64	3	1	1	12,288
Conv 5	64	64	64	64	3	1	1	12,288
Conv 5 -Res	-	-	-	-	-	-	-	-
Conv 6	64	64	64	64	3	1	1	12,288
Conv 7	64	64	64	64	3	1	1	12,288
Conv 7 -Res	-	-	-	-	-	-	-	-
Conv 8	64	32	64	128	3	2	1	73,738
Conv 9	32	32	128	128	3	1	1	147,456
Conv 9 -Res	64	32	64	128	1	2	0	8,192
Conv 10	32	32	128	128	3	1	1	147,456
Conv 11	32	32	128	128	3	1	1	147,456
Conv 11-Res	-	-	-	-	-	-	-	-
Conv 12	32	32	128	128	3	1	1	147,456
Conv 13	32	32	128	128	3	1	1	147,456
Conv 13-Res	-	-	-	-	-	-	-	-
Conv 14	32	32	128	128	3	1	1	147,456
Conv 15	32	32	128	128	3	1	1	147,456
Conv 15-Res	-	-	-	-	-	-	-	-
Conv 16	32	16	128	256	3	2	1	294,912
Conv 17	16	16	256	256	3	1	1	589,824
Conv 17-Res	32	16	126	256	1	2	0	32,256

Conv 18	16	16	256	256	3	1	1	589,824
Conv 19	16	16	256	256	3	1	1	589,824
Conv 19-Res	-	-	-	-	-	-	-	-
Conv 20	16	16	256	256	3	1	1	589,824
Conv 21	16	16	256	256	3	1	1	589,824
Conv 21-Res	-	-	-	-	-	-	-	-
Conv 22	16	16	256	256	3	1	1	589,824
Conv 23	16	16	256	256	3	1	1	589,824
Conv 23-Res	-	-	-	-	-	-	-	-
Conv 24	16	16	256	256	3	1	1	589,824
Conv 25	16	16	256	256	3	1	1	589,824
Conv 25-Res	-	-	-	-	-	-	-	-
Conv 26	16	16	256	256	3	1	1	589,824
Conv 27	16	16	256	256	3	1	1	589,824
Conv 27-Res	-	-	-	-	-	-	-	-
Conv 28	16	8	256	512	3	2	0	1,179,648
Conv 29	8	8	512	512	3	1	1	2,359,296
Conv 29-Res	16	8	256	512	1	2	0	131,072
Conv 30	8	8	512	512	3	1	1	2,359,296
Conv 31	8	8	512	512	3	1	1	2,359,296
Conv 31-Res	-	-	-	-	-	-	-	-
Conv 32	8	8	512	512	3	1	1	2,359,296
Conv 33	8	8	512	512	3	1	1	2,359,296
Conv 33-Res	-	-	-	-	-	-	-	-
Avg Pool	8	1	512	512	8	1	0	-
Fully connected layer with bias , Input: 512, hidden units: 1000, activation: Relu								513,000
Fully connected layer with bias, Input: 1000, hidden units : 2								2,002
Total parameters								21,625,860

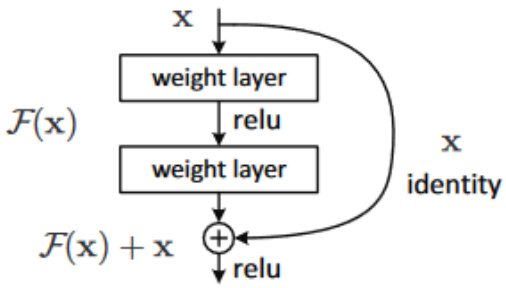


Figure 3.1 : A residual layer (taken from ResNet paper - https://arxiv.org/pdf/1512.03385.pdf)

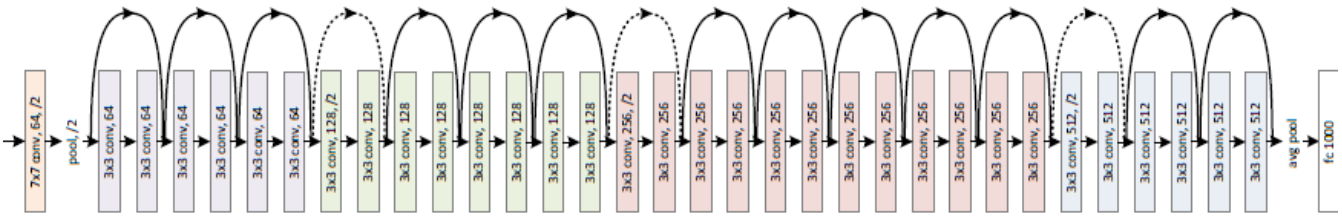


Figure 3.2 : Architecture of ResNet model implementated (taken from ResNet paper - https://arxiv.org/pdf/1512.03385.pdf)

2. Training of the model

Regarding the backpropagation for training, the training dataset was first split into 80% train samples and 10% validation samples using the script generateData.py. Found here: https://github.com/jonPlante/IFT6135_Ass1

Online data augmentation methods where used to train the model. At the beginning of each epoch, the original training data examples were augmented by 10 pictures. The first five augmented picture were generated using pyTorch (figure 3.3).

```
torch_transform = torchvision.transforms.Compose([
    torchvision.transforms.ColorJitter(hue=.05, saturation=.05),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomRotation(20),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomVerticalFlip(),
    torchvision.transforms.RandomGrayscale(),
    torchvision.transforms.RandomAffine(20, translate=[0.1,0.1], scale=None, shear=5, resample=False, fillcolor=0),
    torchvision.transforms.RandomResizedCrop(64, scale=(0.96, 1.0), ratio=(0.95, 1.05)),
    torchvision.transforms.ToTensor(),
])
```

Figure 3.3: Image augmentation techniques used from pyTorch

The second five augmented images pictures were generated using the image augment module since this one can blur (Gaussian blur) the picture and add noise (Dropout) to it (figure 3.4).

```
class ImgAugTransform:
  def __init__(self):
    self.aug = iaa.Sequential([
        iaa.Scale((64, 64)),
        iaa.Sometimes(0.6, iaa.GaussianBlur(sigma=(0, 2.0))),
        iaa.Fliplr(0.5),
        iaa.Affine(rotate=(-20, 20), mode='symmetric'),
        iaa.Sometimes(0.6,
                      iaa.OneOf([iaa.Dropout(p=(0, 0.1)),
                                 iaa.CoarseDropout(0.1, size_percent=0.7)])),
        iaa.AddToHueAndSaturation(value=(-10, 10), per_channel=True)
    ])

  def __call__(self, img):
    img = np.array(img)
    return self.aug.augment_image(img)
```

Figure 3.4: Image augmentation techniques used from Image Augmentation module

We then trained the model with a cross entropy loss function and stochastic gradient descent. It was trained for 31 epochs (limit of time) with a learning rate of 0.05 for the first 25 epochs, 0.01 for the next 3 and 0.001 for the next 3. Mini batch for every epoch was 50.

3. Training vs validation

Figure 3.5 and 3.6 show the training and validation errors and training and validation losses.

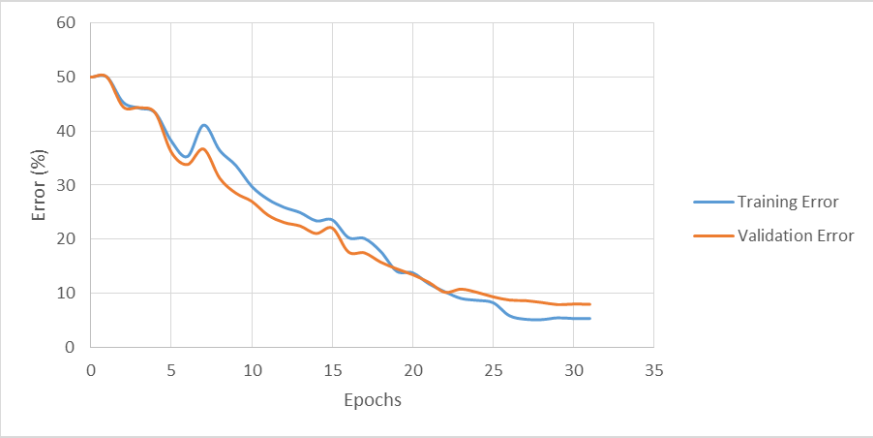


Figure 3.5: Training and validation errors as of function of number of epochs

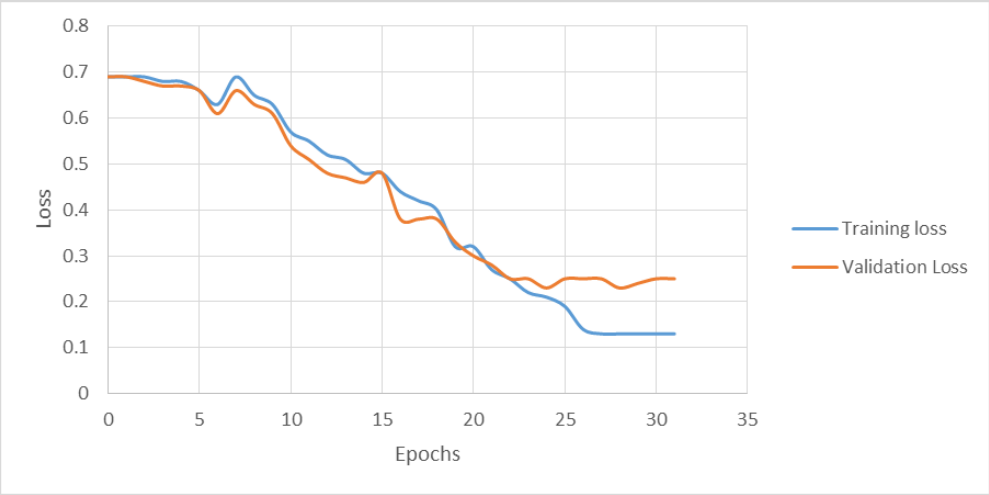


Figure 3.6: Training and validation losses as of function of number of epochs

After 31 epochs, the training error was 5.33%, the validation error was 7.95% and test error (obtained from kaggle – Pos 19: Jonathan) was to be around 9%. As one observes figure 3.6, one can see that the training and validation loss drops and follows each other closely until epoch 25. At epoch 25, we see that the training loss starts pulling away from the validation loss. This is also where the learning rate was decreased. This was done because with a constant learning rate of 0.05, after 25 epochs, the model started overfitting the examples (train set error decreasing while validation error increased). It was then decided to continue training the model from epoch 25 but with a smaller learning rates.

When observing figure 3.5, we see that the error for training and validation drops at the same rate until epoch 25. After epoch 25, the validation error drops slower. This makes sense since the loss on the validation set starts diverging from the training loss. With more epochs we expect to see this divergence grow, and the error on the validation grow. The model generalized well since the error on the validation set was 7.95% and the error on the test set was of about 9%.

To improve our performance we can train for more epochs to see if there is an improvement. We could also use gradient descent with momentum since this would help converge more rapidly to an optimal solution. We could also use weight decay to make sure our weights are not getting too big and avoid having one set of weights dominating the other sets. We could also use batch norm since it was shown to improve significantly the performances of CNNs.

4. Improvements and tries

The first try done corresponded to a model with 3 Conv layers and two fully connected layer, MB of 50, LR=0.01, activation: RELU, and 30 epochs of training with no data augmentation (table 3.2). The validation accuracy was 74%.

Table 3.2: Detailed architecture of convolutional neural network classifier try 1 (dilution=1)

Layer	Input Size	Output Size	Input channels	Output channels	Mask size	stride	Pad	Parameters
Conv 1	64	64	3	32	5	1	2	2400
Conv 2	64	64	32	32	5	1	2	25,600
Conv 3	64	64	32	32	5	1	2	25,600
Max Pool 1	64	32	32	32	2	2	0	-
Fully connected layer with bias , Input: 32768, hidden units: 512, activation: Relu								16.8 M
Fully connected layer with bias, Input: 512, hidden units : 2								1,026
Total parameters								16.9 M

The second try done corresponded a model with 3 conv layers and 2 fully connected layer, MB of 50, LR=0.1, activation: RELU and 30 epochs of training with no data augmentation (table 3.3). After each of the first 2 conv layer there was a pooling layer. The validation accuracy was 74.5%.

Table 3.3: Detailed architecture of convolutional neural network classifier try 2 (dilution=1)

Layer	Input Size	Output Size	Input channels	Output channels	Mask size	stride	Pad	Parameters
Conv 1	64	64	3	64	3	1	1	1728
Max Pool 1	64	32	64	64	2	2	0	-
Conv 2	32	64	64	128	3	1	1	73,728
Max Pool 2	32	16	128	128	2	2	0	-
Conv 3	16	16	128	256	3	1	1	294,912
Fully connected layer with bias , Input: 65536, hidden units: 1024, activation: Relu								67.1 M

Fully connected layer with bias, Input: 1024, hidden units : 2	2050
Total parameters	67.8 M

The third try done corresponded a model with 9 conv layers and 3 fully connected layer, MB of 50, LR=0.1, activation: RELU and 30 epochs of training with no data augmentation (table 3.4). After every 3 conv layer there is a pooling layer. The validation accuracy was 56.5%.

Table 3.4: Detailed architecture of convolutional neural network classifier try 3 (dilution=1)

Layer	Input Size	Output Size	Input channels	Output channels	Mask size	stride	Pad	Parameters
Conv 1	64	64	3	64	3	1	1	1728
Conv 2	64	64	64	64	3	1	1	36,864
Conv 3	64	64	64	64	3	1	1	36,864
Max Pool 1	64	32	64	64	2	2	0	-
Conv 4	32	32	64	128	3	1	1	73,728
Conv 5	32	32	128	128	3	1	1	147,456
Conv 6	32	32	128	128	3	1	1	147,456
Max Pool 2	32	16	128	128	2	2	0	-
Conv 7	16	16	128	256	3	1	1	294,912
Conv 8	16	16	256	256	3	1	1	589,824
Conv 9	16	16	256	256	3	1	1	589,824
Max Pool 3	16	8	256	256	2	2	0	-
Fully connected layer with bias , Input: 16384, hidden units: 1024, activation: Relu								16.7 M
Fully connected layer with bias , Input: 1024, hidden units: 2048, activation: Relu								2.1 M
Fully connected layer with bias, Input: 2048, hidden units : 2								4098
Total parameters								18.9 M

The third try done was deeper than the other tries, but did not perform well. It was determined that there was most likely a vanishing gradient issue. The fourth try corresponded to the third try but with residual connections (ResNet style) (table 3.5). The validation accuracy was 79.3%. There was no vanishing gradient issue now.

Table 3.5: Detailed architecture of convolutional neural network classifier try 4 (dilution=1)

Layer	Input Size	Output Size	Input channels	Output channels	Mask size	stride	Pad	Parameters
Conv 1	64	64	3	64	3	1	1	1728
Conv 2	64	64	64	64	3	1	1	36,864
Conv 3	64	64	64	64	3	1	1	36,864
Conv 3 –Res	-	-	-	-	-	-	-	-
Max Pool 1	64	32	64	64	2	2	0	-
Conv 4	32	32	64	128	3	1	1	73,728
Conv 5	32	32	128	128	3	1	1	147,456
Conv 6	32	32	128	128	3	1	1	147,456

Conv 6 –Res	-	-	-	-	-	-	-	-
Max Pool 2	32	16	128	128	2	2	0	-
Conv 7	16	16	128	256	3	1	1	294,912
Conv 8	16	16	256	256	3	1	1	589,824
Conv 9	16	16	256	256	3	1	1	589,824
Conv 9 –Res	-	-	-	-	-	-	-	-
Max Pool 3	16	8	256	256	2	2	0	-
Fully connected layer with bias , Input: 16384, hidden units: 1024, activation: Relu								16.7 M
Fully connected layer with bias , Input: 1024, hidden units: 2048, activation: Relu								2.1 M
Fully connected layer with bias, Input: 2048, hidden units : 2								4098
Total parameters								18.9 M

The fifth try corresponded to the final architecture of table 3.1. Different hyper parameters were used and the following results can be seen in table 3.6.

Table 3.6: Hyper parameter search of final architecture

Try	Online Data Augmentation	Epoch (e)	Learning rate	Mini batch size	Accuracy on validation set (%)
1	No	30	0.05	75	81.03
2	No	30	0.01	50	82.36
3	No	30	0.05	50	83.23
4	Yes	30	0.05	50	89.89
5	Yes	30	$e \leq 25$: 0.05 $25 < e \leq 28$: 0.01 $28 < e$: 0.001	50	92.10

We can see that the biggest improvement is online data augmentation. This makes sense since the model has seen more unique examples so therefore is able to generalize better. Thus, one option would be to add more data to generalize better while keeping the online augmentation technique. We also see that a decaying learning rate has helped improved the results. We should explore that avenue as well to improve our results. We can also use techniques that were not allowed for this assignment, such as momentum and batch norm. We could also train for more epochs to see if it can continue improving. For this assignment we restricted ourselves to 30 epochs for time constraints.

Misclassified pictures:

Figure 3.7 shows images that were misclassified or had predictions of about 50% for each class in the validation set.

Figure 3.7: Images that were misclassified or were close to 50% prediction. First 3 are cats, last 2 are dogs

One can see when looking at figure 3.7 that some of the images misclassified by the model are even ambiguous for a human. When we look at the 3rd image (cat), we can see a white animal but it is hard to decipher if it is a dog or cat. Usually a big difference between cats and dogs are the ears. Cat ears are generally pointier. When we look at the first misclassified image, we notice the cat ears are cut off. The absence of this feature may explain why the CNN had trouble. When we look at the 4th image, the dog has pointy ears. This is probably why it was classified as cat.

Kernels:

Here we will show some 3 kernels of the first layer and their effects. They mostly detect edge and perform segmentation.

Initial image: 

Kernel 1: A type of horizontal edge detector

```
[-0.07159605 -0.30559132 -0.2743282 ]
  [-0.29949027 -0.3262786  -0.14048253]
  [-0.16235012 -0.37632424 -0.35723722]]

 [[-0.09377994  0.13382609 -0.05864702]
  [-0.19510555 -0.176316    0.09913731]
  [ 0.049535    0.01224533  0.13678668]]

 [[ 0.24271184  0.46160275  0.42395422]
  [ 0.4245433   0.25482687  0.4972244 ]
  [ 0.09067275  0.45930833  0.39013326]]
```

Kernel 2: Some of sort contrast detector and segmentor

```
[[[-1.4317319e-01  9.2344254e-02 -7.4447826e-02]
  [ 6.5906668e-01 -3.5712127e-02 -4.5690153e-02]
  [-1.8511701e-01  4.9732620e-04  9.6774250e-02]]

 [[ 3.0953391e-02 -2.9383725e-01  1.3376038e-01]
  [ 6.6162133e-01 -3.8748018e-02 -1.8863273e-01]
  [-1.5394519e-01 -2.0559652e-01 -1.0014882e-01]]

 [[ 5.4509167e-02 -1.3220550e-01 -1.6021746e-01]
  [ 6.9618732e-01  1.0120367e-01  1.7610996e-04]
  [-3.6030050e-02 -8.8958137e-02 -1.2735005e-02]]]
```

Kernel 3: Diagonal edge detector

```
[[[-0.20861651 -0.12034151 -0.24971011]
  [-0.7396918  -0.26840076 -0.05806274]
  [ 0.10873225 -0.1842792  -0.13628414]]

 [[ 0.06545036  0.18466534  0.23824137]
  [-0.5213606   0.22328472  0.15282047]
  [ 0.04469083  0.06790651  0.22750814]]

 [[-0.02449681  0.12807378 -0.10263061]
  [-0.2072013   0.21870361  0.34698924]
  [ 0.01057074  0.3294812   0.22669701]]]
```

Conclusion:

We have observed that the biggest improvements was having residual connections and allowing gradient to flow for training. We also observed that data augmentation played a huge part in improving our model. It is therefore recommended to add more data to the training (with new images, or more image augmentation) and to possibly have more layers as we see that the first layers detect edges and contrasts and the next layers detect more sophisticated features. We also think that in the list of methods that were allowed for this assignment but that were not implemented, a decaying learning rate would be worth testing, since when we used a LR that changed after 25 epochs it did improve our results.

The next pages show the code used.

Problem 3

February 17, 2019

https://colab.research.google.com/drive/1aSECHvD_jfC-6ErJMXUFT6RMJFiYyUfJ

You must use the data generation script: https://github.com/jonPlante/IFT6135_Ass1/blob/master/generat to prepare the data

Then save on your drive as catDog.pkl.gz

```
In [ ]: !pip3 install 'torch==0.4.0'
        !pip3 install 'torchvision==0.2.1'
        !pip3 install --no-cache-dir -I 'pillow==5.1.0'

        # Restart Kernel
        # This workaround is needed to properly upgrade PIL on Google Colab.
        import os
        os._exit(00)
```

```
In [ ]: !pip install git+https://github.com/aleju/imgaug
        from imgaug import augmenters as iaa
        import imgaug as ia
```

```
In [ ]: import numpy as np
        import gzip
        import pickle
        import matplotlib.pyplot as plt
        from google.colab import drive
        drive.mount('/content/gdrive')
```

```
In [ ]: with gzip.open('/content/gdrive/My Drive/catDog.pkl.gz', 'rb') as f:
          train_set, valid_set, test_set = pickle.load(f, encoding='latin1')
```

```
In [ ]: import torch
        import PIL
        from PIL import Image
        import torchvision
        import torchvision.transforms as transforms
        from torch.autograd import Variable
        from torch.utils.data.dataset import Dataset
```

```
In [ ]: torch_transform = torchvision.transforms.Compose([
            torchvision.transforms.ColorJitter(hue=.05, saturation=.05),
```

```
            torchvision.transforms.RandomHorizontalFlip(),
            torchvision.transforms.RandomRotation(20),
            torchvision.transforms.RandomHorizontalFlip(),
            torchvision.transforms.RandomVerticalFlip(),
            torchvision.transforms.RandomGrayscale(),
            torchvision.transforms.RandomAffine(20, translate=[0.1,0.1], scale=None, shear=5, r
            torchvision.transforms.RandomResizedCrop(64, scale=(0.96, 1.0), ratio=(0.95, 1.05))
            torchvision.transforms.ToTensor(),
        ])
```

In [ ]:
```
class ImgAugTransform:
  def __init__(self):
    self.aug = iaa.Sequential([
        iaa.Scale((64, 64)),
        iaa.Sometimes(0.6, iaa.GaussianBlur(sigma=(0, 2.0))),
        iaa.Fliplr(0.5),
        iaa.Affine(rotate=(-20, 20), mode='symmetric'),
        iaa.Sometimes(0.6,
                      iaa.OneOf([iaa.Dropout(p=(0, 0.1)),
                                 iaa.CoarseDropout(0.1, size_percent=0.7)])),
        iaa.AddToHueAndSaturation(value=(-10, 10), per_channel=True)
    ])

  def __call__(self, img):
    img = np.array(img)
    return self.aug.augment_image(img)

Img_Aug_T = ImgAugTransform()
```

In [ ]:
```
import random
random.shuffle(train_set)
train=[]
valid=[]
test=[]
for i in range(len(train_set)):
  x=torch.Tensor(train_set[i][0])
  train.append([x,train_set[i][1]])
  #data augmentation
  for j in range(5):
    #pytorch
    x=torch_transform(Image.fromarray(np.uint8(np.moveaxis(train_set[i][0],0,2)*255)))
    train.append([x,train_set[i][1]])
    #dataAugment
    x=Img_Aug_T(np.uint8(np.moveaxis(train_set[i][0],0,2)*255))
    x=torch.Tensor(np.moveaxis(x/255,2,0))
    train.append([x,train_set[i][1]])
  train_set[i]=[]
  if i%500==0 and not i==0:
```

```
            print('\t'+str(np.round(i/len(train_set)*100,2))+'% complete')
        train_set=[]
        for i in range(len(valid_set)):
          x=torch.Tensor(valid_set[i][0])
          valid.append([x,valid_set[i][1]])
          valid_set[i]=[]
        valid_set=[]
        for i in range(len(test_set)):
          x=torch.Tensor(test_set[i][0])
          test.append([x,test_set[i][1]])
          test_set[i]=[]
        test_set=[]
```

In [ ]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3,padding=1)

        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3,padding=1,stride=2)
        self.conv4 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv5 = nn.Conv2d(64, 128, 1,padding=0,stride=2)
        self.conv6 = nn.Conv2d(128, 128, 3,padding=1)
        self.conv7 = nn.Conv2d(128, 256, 3,padding=1,stride=2)
        self.conv8 = nn.Conv2d(256, 256, 3,padding=1)
        self.conv9 = nn.Conv2d(128, 256, 1,padding=0,stride=2)
        self.conv10 = nn.Conv2d(256, 512, 3,padding=1,stride=2)
        self.conv11 = nn.Conv2d(512, 512, 3,padding=1)
        self.conv12 = nn.Conv2d(256, 512, 1,padding=0,stride=2)

        self.fc1 = nn.Linear(512*1*1,1000)
        self.fc2 = nn.Linear(1000, 2)
        self.AvgPool = nn.AvgPool2d(8,1)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        residual_1=x

        x = F.relu(self.conv2(x))
        x = F.relu(self.conv2(x)+residual_1)
        residual_2=x
```

```
x = F.relu(self.conv2(x))
x = F.relu(self.conv2(x)+residual_2)
residual_3=x

x = F.relu(self.conv2(x))
x = F.relu(self.conv2(x)+residual_3)
residual_4=x

x = F.relu(self.conv3(x))
x = F.relu(self.conv4(x)+self.conv5(residual_4))
residual_5=x

x = F.relu(self.conv6(x))
x = F.relu(self.conv6(x)+residual_5)
residual_6=x

x = F.relu(self.conv6(x))
x = F.relu(self.conv6(x)+residual_6)
residual_7=x

x = F.relu(self.conv6(x))
x = F.relu(self.conv6(x)+residual_7)
residual_8=x

x = F.relu(self.conv7(x))
x = F.relu(self.conv8(x)+self.conv9(residual_8))
residual_9=x

x = F.relu(self.conv8(x))
x = F.relu(self.conv8(x)+residual_9)
residual_10=x

x = F.relu(self.conv8(x))
x = F.relu(self.conv8(x)+residual_10)
residual_11=x

x = F.relu(self.conv8(x))
x = F.relu(self.conv8(x)+residual_11)
residual_12=x

x = F.relu(self.conv8(x))
x = F.relu(self.conv8(x)+residual_12)
residual_13=x

x = F.relu(self.conv8(x))
x = F.relu(self.conv8(x)+residual_13)
residual_14=x
```

```
        x = F.relu(self.conv10(x))
        x = F.relu(self.conv11(x)+self.conv12(residual_14))
        residual_15=x

        x = F.relu(self.conv11(x))
        x = F.relu(self.conv11(x)+residual_15)
        residual_16=x

        x = F.relu(self.conv11(x))
        x = F.relu(self.conv11(x)+residual_16)
        x=self.AvgPool(x)

        x = x.view(-1, 512*1*1)
        x = F.relu(self.fc1(x))

        x = self.fc2(x)


        return x
```

In [ ]:
```
epochs=60
start=31
MB=50
LR=0.001
use_cuda = True

numTrain=len(train)
numValid=len(valid)




net = Net()
#if want to load modelcomment
#net.load_state_dict(torch.load('/content/gdrive/My Drive/model_'+str(start)+'.pkl'))


if use_cuda and torch.cuda.is_available():
    net.cuda()
    print('using cuda')
else:
  print('not using cuda')

trainloader = torch.utils.data.DataLoader(train, batch_size=MB,shuffle=False, num_worke
validloader = torch.utils.data.DataLoader(valid, batch_size=MB,shuffle=False, num_worke


criterion = nn.CrossEntropyLoss()#nn.BCELoss()
```

```
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.0)

train_data=np.zeros((epochs+1,3))
valid_data=np.zeros((epochs+1,3))

print('Calculating initial training loss')
total_loss = 0.0
total_error=0
for i, data in enumerate(trainloader, 0):
 # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)#.type(torch.FloatTensor))

  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()

  outputs = net(images)
  loss = criterion(outputs, labels)
  total_error+=np.sum(np.abs(np.argmax(outputs.data.cpu().numpy(),axis=1)-labels.cpu()

  total_loss+=loss.data.cpu()
  if i*MB%10000==0 and not i==0:
    print('\t'+str(np.round(i*MB/numTrain*100,2))+'% complete')


train_data[0,:]=[0,total_loss/(numTrain/MB),total_error/numTrain]
print('Initial training loss: ' +str(np.round(train_data[0,1],2))+', training error: '


print('Calculating initial validation loss')
total_loss = 0.0
total_error=0
for i, data in enumerate(validloader, 0):
  # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)#.type(torch.FloatTensor))

  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()

  outputs = net(images)
  loss = criterion(outputs, labels)
  total_error+=np.sum(np.abs(np.argmax(outputs.data.cpu().numpy(),axis=1)-labels.cpu()
  total_loss+=loss.data.cpu()
```

```
valid_data[0,:]=[0,total_loss/(numValid/MB),total_error/numValid]
print('Initial validation loss: ' +str(np.round(valid_data[0,1],2))+', validation erro


for epoch in range(start,epochs):  # loop over the dataset multiple times

  trainloader=[]
  validloader=[]
  train=[]
  valid=[]
  test=[]

  with gzip.open('/content/gdrive/My Drive/catDog.pkl.gz', 'rb') as f:
    train_set, valid_set, test_set = pickle.load(f, encoding='latin1')

  random.shuffle(train_set)

  for i in range(len(train_set)):
    x=torch.Tensor(train_set[i][0])
    train.append([x,train_set[i][1]])
     #data augmentation
    for j in range(3):
      x=torch_transform(Image.fromarray(np.uint8(np.moveaxis(train_set[i][0],0,2)*255))
      train.append([x,train_set[i][1]])
      x=Img_Aug_T(np.uint8(np.moveaxis(train_set[i][0],0,2)*255))
      x=torch.Tensor(np.moveaxis(x/255,2,0))
      train.append([x,train_set[i][1]])
    train_set[i]=[]
    if i%500==0 and not i==0:
      print('\t'+str(np.round(i/len(train_set)*100,2))+'% complete')
  train_set=[]
  for i in range(len(valid_set)):
    x=torch.Tensor(valid_set[i][0])
    valid.append([x,valid_set[i][1]])
    valid_set[i]=[]
  valid_set=[]
  for i in range(len(test_set)):
    x=torch.Tensor(test_set[i][0])
    test.append([x,test_set[i][1]])
    test_set[i]=[]
  test_set=[]

  trainloader = torch.utils.data.DataLoader(train, batch_size=MB,shuffle=False, num_wo
  validloader = torch.utils.data.DataLoader(valid, batch_size=MB,shuffle=False, num_wo

  running_loss = 0.0
  num_ex=0
```

```
for i, data in enumerate(trainloader, 0):
  # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)#.type(torch.FloatTensor))

  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()

  # zero the parameter gradients
  optimizer.zero_grad()

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

  # print statistics
  running_loss += loss.data.cpu().numpy()
  num_ex+=1
  if i*MB%10000 == 0 and not i==0:
    print('\tLoss: '+str(np.round(running_loss/num_ex,3))+', '+str(np.round(i*MB/num
    num_ex=0
    running_loss=0
print('Epoch '+str(epoch+1) + ' complete')

torch.save(net.state_dict(), '/content/gdrive/My Drive/model_'+str(epoch+1)+'.pkl')

print('Calculating training loss')
total_loss = 0.0
total_error=0
for i, data in enumerate(trainloader, 0):
  # get the inputs
  images, labels = data
  images=Variable(images)
  labels=Variable(labels)#.type(torch.FloatTensor))

  if use_cuda and torch.cuda.is_available():
    images = images.cuda()
    labels = labels.cuda()

  outputs = net(images)
  loss = criterion(outputs, labels)
  total_error+=np.sum(np.abs(np.argmax(outputs.data.cpu().numpy(),axis=1)-labels.cpu
  total_loss+=loss.data.cpu()
```

```
        if i*MB%10000==0 and not i==0:
          print('\t'+str(np.round(i*MB/numTrain*100,2))+'% complete')

      train_data[epoch+1,:]=[epoch+1,total_loss/(numTrain/MB),total_error/numTrain]
      print('Training loss: ' +str(np.round(train_data[epoch+1,1],2))+', training error: '+

      print('Calculating validation loss')
      total_loss = 0.0
      total_error=0
      for i, data in enumerate(validloader, 0):
        # get the inputs
        images, labels = data
        images=Variable(images)
        labels=Variable(labels)#.type(torch.FloatTensor)

        if use_cuda and torch.cuda.is_available():
          images = images.cuda()
          labels = labels.cuda()

        outputs = net(images)
        loss = criterion(outputs, labels)
        total_error+=np.sum(np.abs(np.argmax(outputs.data.cpu().numpy(),axis=1)-labels.cpu
        total_loss+=loss.data.cpu()

      valid_data[epoch+1,:]=[epoch+1,total_loss/(numValid/MB),total_error/numValid]
      print('Validation loss: ' +str(np.round(valid_data[epoch+1,1],2))+', validation erro

    print('Finished Training')
    torch.cuda.empty_cache()


In [ ]: # if want to load model uncomment

    #net = Net()
    #net.load_state_dict(torch.load('/content/gdrive/My Drive/model_'+str(31)+'.pkl'))
    #use_cuda=True
    #
    #if use_cuda and torch.cuda.is_available():
    #    net.cuda()
    #    print('using cuda')
    #else:
    #  print('not using cuda')
    #

    print('Calculating performance on test set')
    testloader = torch.utils.data.DataLoader(test, batch_size=1,shuffle=False, num_workers=
    test_data=np.zeros((len(test),2))
    for i, data in enumerate(testloader, 0):
```

```
        # get the inputs
          images, labels = data
          images=Variable(images)
          labels=Variable(labels[0])

          if use_cuda and torch.cuda.is_available():
            images = images.cuda()
            labels = labels.cuda()

          outputs = net(images)
          #print(labels.cpu().numpy())
          test_data[i,:]=[labels.cpu().numpy(),np.argmax(outputs.data.cpu().numpy())]
        test_data=test_data[np.argsort(test_data[:, 0])]

        print('done')
```

In [ ]:
```
        import csv
        with open('/content/gdrive/My Drive/submission_file.csv','w') as csvFile:
            writer = csv.writer(csvFile,lineterminator = '\n')
            writer.writerow(['id','label'])
            for i in range(test_data.shape[0]):
              if test_data[i,1]==0:
                writer.writerow([test_data[i,0].astype(np.int32),'Cat'])
              else:
                writer.writerow([test_data[i,0].astype(np.int32),'Dog'])
```