In this lab we will go over over basic stochastic optimization and uses in pytorch

(1) Setup the MNIST dataloaders for both the training (and now as well test) set as in Lab 1 part 2. You do not need to iterate through the dataloaders, these will be used in the rest of the lab.

In [41]:

```
import numpy as np
import math
import matplotlib.pyplot as plt
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import timeit
```

In [42]:

```
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

In [43]:

```
batch_size_train = 256
batch_size_test = 256
```

In [44]:

```
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=batch_size_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=False, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=batch_size_test, shuffle=True)
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
print(example_data.shape)
```

```
torch.Size([256, 1, 28, 28])
```

(2) Consider your work from [Lab 1 part 2 (2)]. Modify the neural network as follows. The hidden outputs should be size 100 in both hidden layers and the activations should be relu. Modify the final layer to output 10 values instead of 1.

Your code should implement $f(x) = W_2\rho(W_1\rho(W_0x + b_0) + b_1) + b_2$ with $f : R^{786} - > R^{10}$ and $\rho = relu$

Initialize the weights using a variant of xavier intialization $w_{ij} \sim N(0, \frac{1}{\sqrt{n_i}})$ where n_i is the size of the layer input. Initialize the biases as 0. Write a helper function to perform this initialization for subsequent parts of this lab

In [50]:

```
def xavier(m,h):
    return np.random.randn(m,h)*np.math.sqrt(1/m)
xavier(100,100)
```

Out[50]:

```
array([[-0.04321282, -0.10218911,  0.01614617, ...,  0.16381488,
         0.14274758,  0.01754349],
       [ 0.07148129, -0.14143608,  0.12969472, ...,  0.02727722,
         0.0792694 ,  0.20976034],
       [-0.08781296,  0.0210077 , -0.03612951, ...,  0.02669059,
         0.02558466,  0.14200836],
       ...,
       [ 0.0617158 ,  0.01722537, -0.11475476, ..., -0.27288359,
        -0.15668369, -0.20346779],
       [ 0.12547211, -0.08124148, -0.00585535, ...,  0.20889572,
         0.08821267,  0.08180942],
       [ 0.13761689,  0.02925177, -0.05195731, ..., -0.03895996,
         0.0825273 ,  0.05838457]])
```

In [51]:

```
param_dict={"W0":xavier(784, 100),
    # print(W1)
    "W1": xavier(100, 100), #torch.randn((12, 10), requires_grad=True)
    "W2": xavier(100, 10), #torch.randn((10, 1), requires_grad=True)
    # define the bias terms
    "B0": np.zeros(100), #torch.randn((12), requires_grad=True)
    "B1": np.zeros(100), #torch.randn((10), requires_grad=True)
    "B2": np.zeros(10), #torch.randn((1), requires_grad=True)]
       }
```

In [161]:

```
## Define the network
def my_nn(input,param_dict):
    W0=param_dict["W0"]
    W1=param_dict["W1"]
    W2=param_dict["W2"]
    B0=param_dict["B0"]
    B1=param_dict["B1"]
    B2=param_dict["B2"]
    features = torch.flatten(input).reshape(-1)
        # calculate hidden and output layers
    h1 = torch.relu_((features @ W0) + B0)
    h2 = torch.relu_((h1 @ W1) + B1)
    output = h2 @ W2 + B2
#     output=(output - 2)/0.5
#     print(output)
#     print(output.reshape(-1,1))
    return F.log_softmax(output.reshape(-1,1), dim=1)

start = timeit.default_timer()
for idx in range(len(example_data)):
     my_nn(example_data[idx][0],param_dict)
stop = timeit.default_timer()
print('FF 256 images Time= :',stop - start)
```

```
FF 256 images Time= : 0.6902508880011737
```

We will evaluate the cross entropy loss and average accuracy on this randomly initialized dataset. Use the torch.nn.functional.cross_entropy() function to compute the loss $\frac{1}{N}\sum_i^N CrossEntropy(f(x_i), y_i)$ and accuracy. Your accuracy should be close to 10% as the network has random weights. Note the pytorch cross_entropy function already applies the softmax operator so you do not need to apply this just feed the model output directly

In [146]:

```
from math import log2
def cross_entropy(p, q):
    return -sum([p[i]*log2(q[i]) for i in range(len(p))])
```

In [160]:

```
from itertools import repeat
cuda = torch.cuda.is_available()
for data, targets in test_loader:
  #move to GPU if available
#   if cuda:
#     data, target = data.to('cuda'), target.to('cuda')
  #compute model output
    print(len(data))
#     for i in range(len(data)):
    for i in range(1):
        out=my_nn(data[i],param_dict)
#         print(len(targets))
  #comptue accuracy for minibatch
  #compute loss for minibatch
        print(out)
        print(targets[i])
        target_flatted=torch.FloatTensor(np.zeros(10))
        target_flatted[targets[i]]=1
#         repeat(targets[i],10)
        print(target_flatted.shape)
        F.cross_entropy(out,target_flatted)
#aggregate loss and accuracy for all test data
```

```
256
tensor([[-0.2821],
        [-0.2607],
        [ 0.1944],
        [-0.3023],
        [-0.2011],
        [ 0.0768],
        [ 0.3979],
        [ 0.9509],
        [-0.5255],
        [ 0.0853]], dtype=torch.float64)
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]], dtype=torch.float64)
tensor(2)
torch.Size([10])

---------------------------------------------------------------------------
-
RuntimeError                              Traceback (most recent call las
t)
<ipython-input-160-483f3997189c> in <module>
     19 #         repeat(targets[i],10)
     20         print(target_flatted.shape)
---> 21         F.cross_entropy(out,target_flatted)
     22 #aggregate loss and accuracy for all test data
     23 

c:\program files\python37\lib\site-packages\torch\nn\functional.py in cros
s_entropy(input, target, weight, size_average, ignore_index, reduce, reduc
tion)
   2420     if size_average is not None or reduce is not None:
   2421         reduction = _Reduction.legacy_get_string(size_average, red
uce)
-> 2422     return nll_loss(log_softmax(input, 1), target, weight, None, i
gnore_index, None, reduction)
   2423 
   2424 

c:\program files\python37\lib\site-packages\torch\nn\functional.py in nll_
loss(input, target, weight, size_average, ignore_index, reduce, reduction)
   2216                          .format(input.size(0), target.size(0)))
   2217     if dim == 2:
-> 2218         ret = torch._C._nn.nll_loss(input, target, weight, _Reduct
ion.get_enum(reduction), ignore_index)
   2219     elif dim == 4:
   2220         ret = torch._C._nn.nll_loss2d(input, target, weight, _Redu
ction.get_enum(reduction), ignore_index)

RuntimeError: expected scalar type Long but found Float
```

(3) (a) Without any use of the torch.optim package implement from scratch mini-batch Stochastic Gradient Descent training to minimize the loss $\frac{1}{N}\sum_i^N CrossEntropy(f(x_i), y_i)$ over the MNIST dataset. Use a minibatch size of 128 and a learning rate of 0.01 and run training for 20 epochs.

You will use torch autograd features (e.g. .backward()) to obtain the gradients given each mini-batch at each parameter and then modify the existing parameters based on this gradient.

Store the losses and training of each minibatch and plot each of these (with iterations (not epochs) as the x-axis). You can optionally smooth out these plots over 20-100 iteration window of your choosing to make them cleaner to read. Compute and report the final test accuracy as well at the end of the 20 epochs.

You should end up with 2 plots and a final test accuracy.

In [ ]:

```
import torch


train_losses = [] # use to append the avg loss for each minibatch
train_accs = [] # use to append the avg acc of minibatch

alpha=0.01


for epoch in range(20):
  for ... #Iterate over dataset
    #Compute the gradient for the minibatch

    #Update the model parameters by w_t-alpha*\grad_{w_t}(f(w_t))

    #The following code will clear the gradient buffers for the next iteration
    for (_,param) in param_dict.items():
      if param.grad is not None:
        param.grad.detach_()
        param.grad.zero_()

    #Update loss and acc tracking

#Plot the train loss and acc

#Evaluate on the test set
```

(b) Modify the above code to perform minibatch SGD with momentum. Use a momentum of $\mu = 0.9$ and learning rate $\alpha = 0.01$. Use the following formulation of momentum:

$g = \nabla_w CE(f(w_t, X), Y)$ gradient estimate with mini-batch

$v_{t+1} = \mu * v_t + g$

$w_{t+1} = w - \alpha * v_{t+1}$

Obtain the same plots as before and a final test accuracy

In [ ]:

```
#Reinitialize your network to random!
initialize_nn(param_dict)

losses = [] # use to append the avg loss for each minibatch
train_acc = [] # use to append the avg acc of minibatch

alpha=0.01
mu=0.9

for epoch in range(20):
  for ... #Iterate over dataset
    #Compute the gradient for the minibatch

    #Update the model parameters as described above

    #The following code will clear the gradient buffers for the next iteration
    for (_,param) in param_dict.items():
      if param.grad is not None:
        param.grad.detach_()
        param.grad.zero_()

    #Update loss and acc tracking

#Plot the train loss and acc

#Evaluate on the test set
```

(4) Repeat (3) but now using the package torch.optim.SGD to perform SGD (a) and SGD with momentum (b). Use the same learning rate in the case of (a) and the same learning rate and momentum in the case of (b)

https://pytorch.org/docs/stable/optim.html?highlight=torch%20optim%20sgd#torch.optim.SGD (https://pytorch.org/docs/stable/optim.html?highlight=torch%20optim%20sgd#torch.optim.SGD)

Plot the same training curves and accuracy curves. Overlay this with their respective implementation from (3). Report the final accuracy.

You should end up with 4 plots (train acc,train loss)x(sgd, sgd+momentum) each containing two curves (your implementation from (3) and the torch.optim result). Due to random initialization and random training order you will not get identical results to (3) but the curves and final accuracies should end up relatively close.

In [ ]:

```
#Answer for SGD
import torch
#Make sure to reinitialize your network to random before starting training
initialize_nn(param_dict)

#optim.SGD takes a list of parameters which you can get from your dictionary as follows
parameter_list = param_dict.values()

optimizer = torch.optim.SGD(parameter_list, lr=alpha)
```

In [ ]:

```
#Answer for SGD+momentum
```