Pytorch with the MNIST Dataset - MINST

```
## import libraries
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision

import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable


print(torch.__version__)
```

```
2.5.0+cu121
```

```
args={}
kwargs={}
args['batch_size']=32
args['test_batch_size']=32
args['epochs']=1  #The number of Epochs is the number of times you go through the full dataset.
args['lr']=0.01 #Learning rate is how fast it will decend.
args['momentum']=0.5 #SGD momentum (default: 0.5) Momentum is a moving average of our gradients (helps to keep direction).

args['seed']=1 #random seed
args['log_interval']=10
args['cuda']=True #if the computer has a GPU, type True, otherwise, False
```

This code is adopted from the pytorch examples repository. It is licensed under BSD 3-Clause "New" or "Revised" License. Source: https://github.com/pytorch/examples/ LICENSE: https://github.com/pytorch/examples/blob/master/LICENSE

Load Dataset

The first step before training the model is to import the data. We will use the MNIST dataset which is like the Hello World dataset of machine learning.

Besides importing the data, we will also do a few more things:

- We will tranform the data into tensors using the `transforms` module
- We will use `DataLoader` to build convenient data loaders or what are referred to as iterators, which makes it easy to efficiently feed data in batches to deep learning models.
- As hinted above, we will also create batches of the data by setting the `batch` parameter inside the data loader. Notice we use batches of `32` in this tutorial but you can change it to `64` if you like. I encourage you to experiment with different batches.

```
## transformations
transform = transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])

## download and load training dataset
trainset = datasets.MNIST(root='../data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=args['batch_size'], shuffle=True, **kwargs)

## download and load testing dataset
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(testset, batch_size=args['test_batch_size'], shuffle=True, **kwargs)
```

Exploring the Data

As a practioner and researcher, I am always spending a bit of time and effort exploring and understanding the dataset. It's fun and this is a good practise to ensure that everything is in order.

Let's check what the train and test dataset contains. I will use `matplotlib` to print out some of the images from our dataset.

```
import matplotlib.pyplot as plt
```

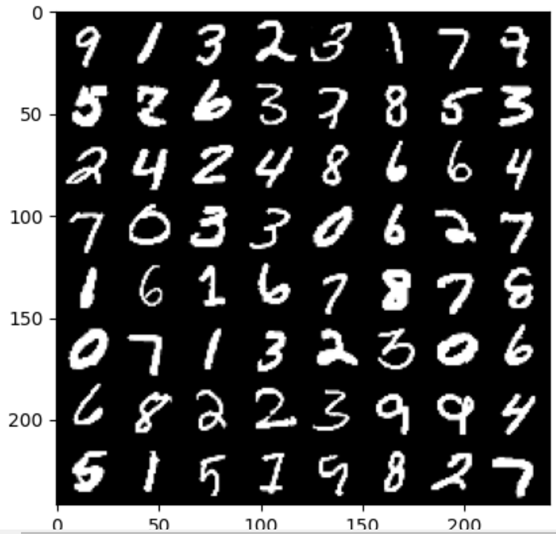
```
import numpy as np

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

## get some random training images
dataiter = iter(train_loader)
images, labels = next(dataiter)

## show images
imshow(torchvision.utils.make_grid(images))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
```



Let's check the dimensions of a batch.

```
for images, labels in train_loader:
    print("Image batch dimensions:", images.shape)
    print("Image label dimensions:", labels.shape)
    break
```

```
Image batch dimensions: torch.Size([64, 1, 28, 28])
Image label dimensions: torch.Size([64])
```

The Model

We provide two fully-connected neural net as the initial architecture.

Here are a few notes for those who are beginning with PyTorch:

- The model below consists of an `__init__()` portion which is where you include the layers and components of the neural network. In our model, we have two fully-connected netork network. We are dealing with an image dataset that is in a grayscale so we only need one channel going in, hence `in_channels=1`.
- After the first layer, we also apply an activation function such as `ReLU`. For prediction purposes, we then apply a `softmax` layer to the last transformation and return the output of that.

```
class Net(nn.Module):
    #This defines the structure of the NN.
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3) #1x28x28 --> 16x26x26
        self.pool = nn.MaxPool2d(2, 2) #16x26x26 --> 16x13x13
        self.fc1 = nn.Linear(16 * 13 * 13, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x=x.view(-1,16 * 13 * 13)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        #Softmax gets probabilities.
        return F.log_softmax(x, dim=1)
```

Now, add one CNN layer with a pooling to the above neural network and rerun the code to see whether you get higher prediction accuracy on the test set.

For example, you may try `out_channels=32`. Kernel size is 5, and for the rest of parameters we use the default values which you can find here.

- In short, the convolutional layer transforms the input data into a specific dimension that has to be considered in the linear layer.

Make sure your flatten the output of CNN layer excluding # of batch so that the input of each example/batch has the same size of the first neural net.

Tips: You can use x.view(-1, # of input size of the first fully-connected layer) or you can use torch.flatten(x, 1).

I always encourage to test the model with 1 batch to ensure that the output dimensions are what we expect.

Start coding or generate with AI.

```
## test the model with 1 batch
model = Net()
#print(model)
for images, labels in train_loader:
    print("batch size:", args['batch_size'])
    out = model(images)
    print(out.shape)
    break
```

```
batch size: 32
torch.Size([64, 10])
```

Training the Model

Now we are ready to train the model.

```
def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args['cuda']:
            data, target = data.cuda(), target.cuda()
        #Variables in Pytorch are differenciable.
        data, target = Variable(data), Variable(target)
        #This will zero out the gradients for this batch.
        optimizer.zero_grad()
        output = model(data)
        # Calculate the loss The negative log likelihood loss. It is useful to train a classification problem with C classes.
        loss = F.nll_loss(output, target)
        #dloss/dx for every Variable
        loss.backward()
        #to do a one-step update on our parameter.
        optimizer.step()
        #Print out the loss periodically.
        if batch_idx % args['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.data.item()))


def test():
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
      for data, target in test_loader:
          if args['cuda']:
              data, target = data.cuda(), target.cuda()
          data, target = Variable(data), Variable(target)
          output = model(data)
          test_loss += F.nll_loss(output, target, size_average=False).data.item() # sum up batch loss
          pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-probability
          correct += pred.eq(target.data.view_as(pred)).long().cpu().sum()

    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)))
```

```
model = Net()
if args['cuda']:
    model.cuda()

optimizer = optim.SGD(model.parameters(), lr=args['lr'], momentum=args['momentum'])

for epoch in range(1, args['epochs'] + 1):
    train(epoch)
    test()
```

```
Train Epoch: 1 [26880/60000 (45%)]      Loss: 0.226806
Train Epoch: 1 [27520/60000 (46%)]      Loss: 0.209637
Train Epoch: 1 [28160/60000 (47%)]      Loss: 0.339335
Train Epoch: 1 [28800/60000 (48%)]      Loss: 0.347948
Train Epoch: 1 [29440/60000 (49%)]      Loss: 0.298498
Train Epoch: 1 [30080/60000 (50%)]      Loss: 0.271223
Train Epoch: 1 [30720/60000 (51%)]      Loss: 0.216389
Train Epoch: 1 [31360/60000 (52%)]      Loss: 0.199735
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.283339
Train Epoch: 1 [32640/60000 (54%)]      Loss: 0.088744
Train Epoch: 1 [33280/60000 (55%)]      Loss: 0.266086
Train Epoch: 1 [33920/60000 (57%)]      Loss: 0.176704
Train Epoch: 1 [34560/60000 (58%)]      Loss: 0.392657
Train Epoch: 1 [35200/60000 (59%)]      Loss: 0.111579
Train Epoch: 1 [35840/60000 (60%)]      Loss: 0.139564
Train Epoch: 1 [36480/60000 (61%)]      Loss: 0.206128
Train Epoch: 1 [37120/60000 (62%)]      Loss: 0.157328
Train Epoch: 1 [37760/60000 (63%)]      Loss: 0.351994
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.415341
Train Epoch: 1 [39040/60000 (65%)]      Loss: 0.338110
Train Epoch: 1 [39680/60000 (66%)]      Loss: 0.407146
Train Epoch: 1 [40320/60000 (67%)]      Loss: 0.425015
Train Epoch: 1 [40960/60000 (68%)]      Loss: 0.305363
Train Epoch: 1 [41600/60000 (69%)]      Loss: 0.472191
Train Epoch: 1 [42240/60000 (70%)]      Loss: 0.340229
Train Epoch: 1 [42880/60000 (71%)]      Loss: 0.244672
Train Epoch: 1 [43520/60000 (72%)]      Loss: 0.160745
Train Epoch: 1 [44160/60000 (74%)]      Loss: 0.268718
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.304855
Train Epoch: 1 [45440/60000 (76%)]      Loss: 0.241072
Train Epoch: 1 [46080/60000 (77%)]      Loss: 0.257948
Train Epoch: 1 [46720/60000 (78%)]      Loss: 0.194221
Train Epoch: 1 [47360/60000 (79%)]      Loss: 0.244654
Train Epoch: 1 [48000/60000 (80%)]      Loss: 0.180502
Train Epoch: 1 [48640/60000 (81%)]      Loss: 0.230826
Train Epoch: 1 [49280/60000 (82%)]      Loss: 0.172106
Train Epoch: 1 [49920/60000 (83%)]      Loss: 0.409677
Train Epoch: 1 [50560/60000 (84%)]      Loss: 0.255971
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.178604
Train Epoch: 1 [51840/60000 (86%)]      Loss: 0.258945
Train Epoch: 1 [52480/60000 (87%)]      Loss: 0.195263
Train Epoch: 1 [53120/60000 (88%)]      Loss: 0.282956
Train Epoch: 1 [53760/60000 (90%)]      Loss: 0.121762
Train Epoch: 1 [54400/60000 (91%)]      Loss: 0.190646
Train Epoch: 1 [55040/60000 (92%)]      Loss: 0.159668
Train Epoch: 1 [55680/60000 (93%)]      Loss: 0.183767
Train Epoch: 1 [56320/60000 (94%)]      Loss: 0.070564
Train Epoch: 1 [56960/60000 (95%)]      Loss: 0.295069
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.303115
Train Epoch: 1 [58240/60000 (97%)]      Loss: 0.196017
Train Epoch: 1 [58880/60000 (98%)]      Loss: 0.198433
Train Epoch: 1 [59520/60000 (99%)]      Loss: 0.110093
/usr/local/lib/python3.10/dist-packages/torch/nn/_reduction.py:51: UserWarning: size_average and reduce args will be deprecated,
  warnings.warn(warning.format(ret))

Test set: Average loss: 0.1699, Accuracy: 9485/10000 (95%)
```

Start coding or generate with AI.