pytorch-mnist-solution

March 27, 2025

1 PyTorch Introduction

Welcome to the introduction of PyTorch. PyTorch is a scientific computing package targeted for two main purposes:

1. A replacement for NumPy with the ability to use the power of GPUs.
2. A deep learning framework that enables the flexible and swift building of neural network models.

Let's get started!

1.0.1 Goals of this tutorial

- Understanding PyTorch's Tensor and neural networks libraries at an overview level.
- Training a neural network using PyTorch.

2 Installing PyTorch

Pytorch provides support for accelerating computation using CUDA enabled GPU's. If your workstation has an NVIDIA GPU, install PyTorch along with the CUDA component.

Install PyTorch and torchvision For this class we will use the current Pytorch version 2.2.2. To install, please uncomment and run the proper line in the upcoming cell depending on your operating system (and CUDA setup). We won't go into details of the installation process.

3 Note!!

All packages should be installed on your conda enviroment. Otherwise, you'd start with mismatching versions loops of differnet libraries, which can make life hard later on when using Python on other projects.

[ ]:
```
# Install a pip package in the current Jupyter kernel
import sys

# TODO: Uncomment or comment the appropriate line for your OS and hardware

# For google colab
```

```
# !python -m pip install torch==2.2.2 torchvision==0.17.2 torchaudio==2.2.2␣
 ↪--index-url https://download.pytorch.org/whl/cu121

# For Linux and probably Windows (CPU)
!{sys.executable} -m pip install torch==2.2.2 torchvision==0.17.2 torchaudio==2.
 ↪2.2 --index-url https://download.pytorch.org/whl/cpu

# For Linux and probably Windows (Prerequisites: Nvidia GPU + CUDA toolkit 11.8)
# !{sys.executable} -m pip install torch==2.2.2+cu118 torchvision==0.17.2+cu118␣
 ↪torchaudio==2.2.2+cu118 --index-url https://download.pytorch.org/whl/cu118


# For OS X/Mac
# !{sys.executable} -m pip install torch==2.2.2 torchvision==0.17.2␣
 ↪torchaudio==2.2.2
```

```
Looking in indexes: https://download.pytorch.org/whl/cpu
Requirement already satisfied: torch==2.2.2 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (2.2.2)
Requirement already satisfied: torchvision==0.17.2 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (0.17.2)
Requirement already satisfied: torchaudio==2.2.2 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (2.2.2)
Requirement already satisfied: filelock in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (4.11.0)
Requirement already satisfied: sympy in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (1.13.2)
Requirement already satisfied: networkx in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (3.4.2)
Requirement already satisfied: jinja2 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (3.1.4)
Requirement already satisfied: fsspec in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torch==2.2.2) (2024.10.0)
Requirement already satisfied: numpy in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torchvision==0.17.2) (1.26.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
torchvision==0.17.2) (10.3.0)
```

```
Requirement already satisfied: MarkupSafe>=2.0 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
jinja2->torch==2.2.2) (2.1.3)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/Users/Francesca/anaconda3/envs/i2dl/lib/python3.11/site-packages (from
sympy->torch==2.2.2) (1.3.0)
```

```
<b>Nvidia GPU</b>
<p>If you have a rather recent Nvidia GPU, you can go ahead and install the CUDA toolkit togetl
<p>There are multiple setups on how to install those on both Linux and Windows, but it depends
But, google or ChatGPT are your new best friends.</p>
<br>
<b>Google Colab Pytorch Installation Time</b>
<p>Google colab might use an older/newer version of pytorch. Since we are mostly using defualt
```

Checking PyTorch Installation and Version

[ ]:
```
import torch
import torchvision
print(f"PyTorch version Installed: {torch.__version__}\nTorchvision version␣
 ↪Installed: {torchvision.__version__}\n")
if not torch.__version__.startswith("2.2"):
    print("you are using an another version of PyTorch. We expect PyTorch 2.2.␣
 ↪You may continue using your version but it"
          " might cause dependency and compatibility issues.")
if not torchvision.__version__.startswith("0.17"):
    print("you are using an another version of torchvision. We expect␣
 ↪torchvision 0.17. You can continue with your version but it"
          " might cause dependency and compatibility issues.")
```

```
PyTorch version Installed: 2.2.2
Torchvision version Installed: 0.17.2
```

That's the end of installation. Let's dive right into PyTorch!

4 Getting Started

In this section you will learn the basic element Tensor and some simple operations in PyTorch. The following block imports the required packages for the rest of the notebook.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

import os
import pandas as pd
pd.options.mode.chained_assignment = None  # default='warn'
```

```
%load_ext autoreload
%autoreload 2
%matplotlib inline

os.environ['KMP_DUPLICATE_LIB_OK']='True' # To prevent the kernel from dying.
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

4.1 1. Tensors (OPTIONAL)

This section is an optional tutorial that covers PyTorch tensors and assumes a basic understanding of numpy arrays. `torch.Tensor` is the central class of PyTorch. Tensors are similar to NumPy's ndarrays. The advantage of using Tensors is that one can easily transfer them from CPU to GPU and therefore computations on tensors can be accelerated with a GPU.

4.2 1.1 Initializing Tensor

Let us construct a NumPy array and a tensor of shape (2,3) directly from data values.

[ ]:
```
# Initializing the Numpy Array
array_np = np.array([[1,2,3],[5,6,7]]) # A NumPy array

# Initializing the Tensor
array_ts = torch.tensor([[1,2,3],[4,5,6]]) # A Tensor

print("Variable array_np:\nDatatype: {}\nShape: {}".format(type(array_np),␣
 ↪array_np.shape))
print("Values:\n", array_np)
print("\n\nVariable array_ts:\nDatatype {}\nShape: {}".format(type(array_ts),␣
 ↪array_ts.shape))
print("Values:\n", array_ts)
```

```
Variable array_np:
Datatype: <class 'numpy.ndarray'>
Shape: (2, 3)
Values:
 [[1 2 3]
 [5 6 7]]


Variable array_ts:
Datatype <class 'torch.Tensor'>
Shape: torch.Size([2, 3])
Values:
 tensor([[1, 2, 3],
        [4, 5, 6]])
```

4.3 1.2 Conversion between NumPy array and Tensor

The conversion between NumPy ndarray and PyTorch tensor is quite easy.

[ ]:
```
# Conversion
array_np = np.array([1, 2, 3])

# Conversion from  a numpy array to a Tensor
array_ts_2 = torch.from_numpy(array_np)

# Conversion from  Tensor to numpy array
array_np_2 = array_ts_2.numpy()

# Change a value of the np_array
array_np_2[1] = -1
array_ts_2[0] = 0

print(f"array_np: {array_np}")
print(f"array_ts_2: {array_ts_2}")
print(f"array_np_2: {array_np_2}")

# Changes in the numpy array will also change the values in the tensor
# AND REVERSED!
assert(array_np[1] == array_np_2[1])
```

```
array_np: [ 0 -1  3]
array_ts_2: tensor([ 0, -1,  3])
array_np_2: [ 0 -1  3]
```

During the conversion, both ndarray and Tensor share the same memory address. Changes in value of one will affect the other.

4.4 1.3 Operations on Tensors

4.4.1 1.3.1 Indexing

We can use the NumPy array-like indexing for Tensors.

[ ]:
```
# Let us take the first two columns from the original tensor array and save it␣
  ↪in a new one
b = array_ts[:2, :2]

# Let's assign the value of first column of the new variable to be zero
b[:, 0] = 0
print(b)
```

```
tensor([[0, 2],
        [0, 5]])
```

We will now select elements which satisfy a particular condition. In this example, let's find those elements of tensor which are array greater than one.

[ ]:
```
# Index of the elements with value greater than one
mask = array_ts > 1
new_array = array_ts[mask]
print(f'array_ts: {array_ts}')
print(f'mask: {mask}')
print(f'new_array: {new_array}')
```

```
array_ts: tensor([[0, 2, 3],
        [0, 5, 6]])
mask: tensor([[False,  True,  True],
        [False,  True,  True]])
new_array: tensor([2, 3, 5, 6])
```

Let's try performing the same operation in a single line of code!

[ ]:
```
c = array_ts[array_ts>1]

# Is the result same as the array from the previous cell?
print(c == new_array)
```

```
tensor([True, True, True, True])
```

4.4.2 1.3.2 Mathematical operations on Tensors

Element-wise operations on Tensors

[ ]:
```
x = torch.tensor([[1,2],[3,4]])
y = torch.tensor([[5,6],[7,8]])

# Elementwise Addition of the tensors
# [[ 6.0  8.0]
#  [10.0 12.0]]

# Addition - Syntax 1
print("x + y: \n{}".format(x + y))

# Addition - Syntax 2
print("x + y: \n{}".format(torch.add(x, y)))

# Addition - Syntax 3
result_add = torch.empty(2, 2)
torch.add(x, y, out=result_add)
print("x + y: \n{}".format(result_add))
```

```
x + y:
tensor([[ 6,  8],
        [10, 12]])
x + y:
tensor([[ 6,  8],
        [10, 12]])
```

```
x + y:
tensor([[ 6.,  8.],
        [10., 12.]])
```

Similar syntax holds for other element-wise operations such as subtraction and multiplication.

When dividing two integers in NumPy as well PyTorch, the result is always a **float**.
For example,

[ ]:
```
x_np = np.array([[1,2],[3,4]])
y_np = np.array([[5,6],[7,8]])
print(x_np / y_np)
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

4.5 1.4 Devices

When training a neural network, it is important to make sure that all the required tensors as well as the model are on the same device. Tensors can be moved between the CPU and GPU using `.to` method.

Let us check if a GPU is available. If it is available, we will assign it to `device` and move the tensor `x` to the GPU.

[ ]:
```
# device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# ADDED mps GPU for M1 mac
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

print(device)

print(f"Original device: {x.device}") # "cpu"

tensor = x.to(device)
print(f"Current device: {tensor.device}") #"cpu" or "cuda" (or "mps")
```

```
mps
Original device: cpu
Current device: mps:0
```

So `x` has been moved on to a CUDA device for those who have a GPU; otherwise it's still on the CPU.

Tip: Try including the .to(device) calls in your codes. It is then easier to port the code to run on a GPU.

5 2. Training a classifier with PyTorch

Now that we are introduced PyTorch tensors, we will look at how to use PyTorch to train neural networks. We will do the following steps:

1. Load data

2. Define a two-layer network
3. Define a loss function and optimizer
4. Train the network
5. Test the network

5.1 2.1 Loading Datasets

The general procedure of loading data is : - Extract data from source - Transform the data into a suitable form (for example, to a Tensor) - Put our data into an object to make it easy to access further on

We will now set our `DataLoader` class to help us to load batches of data.

In PyTorch we can use the `DataLoader` class to accomplish the same objective. It provides more parameters than our `DataLoader` class, such as easy multiprocessing using `num_workers`. You can refer the documentation to learn those additional features.

5.1.1 2.1.2 Torchvision

Specifically for computer vision, the `torchvision` packages has data loaders for many common datasets such as ImageNet, MNIST, Fashion MNIST and additional data transformers for images in `torchvision.datasets` and `torch.utils.data.DataLoader` modules.

This is highly convenient and is useful in avoiding to write boilerplate code.

Let's try loading the MNIST dataset. It has gray-scale images of size $28 * 28$ belonging to 10 different classes of handwritten numbers.

`transforms.Compose` creates a series of transformation to prepare the dataset. - `transforms.ToTensor` convert `PIL image` or numpy.ndarray $(H \times W \times C)$ in the range [0,255] to a `torch.FloatTensor` of shape $(C \times H \times W)$ in the range [0.0, 1.0].

- `transforms.Normalize` normalize a tensor image with the provided mean and standard deviation.

[ ]:
```
# Mean and standard deviations have to be sequences (e.g. tuples),hence we add␣
↪a comma after the values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,))])
```

`datasets.MNIST` downloads the MNIST dataset and transforms it using our previous cell definition. By setting the value of `train`, we get the training and test set.

[ ]:
```
mnist_dataset = torchvision.datasets.MNIST(root='../datasets', train=True,
                                                                download=True,␣
↪transform=transform)
mnist_test_dataset = torchvision.datasets.MNIST(root='../datasets', train=False,
                                                                download=True,␣
↪transform=transform)
```

`torch.utils.data.Dataloader` takes our training data or test data with parameter `batch_size` and `shuffle`. The variable `batch_size` defines how many samples per batch to load. The variable

`shuffle=True` makes the data reshuffled at every epoch.

[ ]:
```
from torch.utils.data import DataLoader

mnist_dataloader = DataLoader(mnist_dataset, batch_size=8)
mnist_test_dataloader = DataLoader(mnist_test_dataset, batch_size=8)

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

Let's look at the first batch of data from the `mnist_dataloader`.

[ ]:
```
# We can use the exact same way to iterate over samples
for i, item in enumerate(mnist_dataloader):
    print('Batch {}'.format(i))
    image, label = item
    print(f"Datatype of Image: {type(image)}")
    print(f"Shape of the Image: {image.shape}")
    print(f"Label Values: {label}")

    if i+1 >= 1:
        break
```

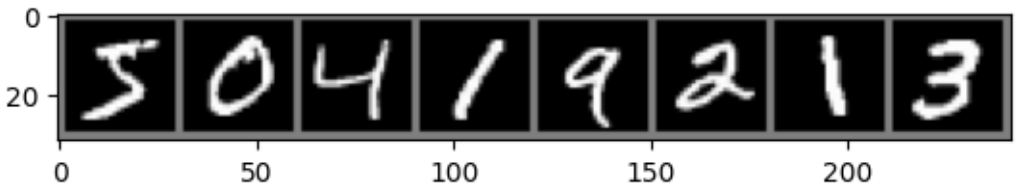
```
Batch 0
Datatype of Image: <class 'torch.Tensor'>
Shape of the Image: torch.Size([8, 1, 28, 28])
Label Values: tensor([5, 0, 4, 1, 9, 2, 1, 3])
```

Since we loaded the data with `batch_size` 8, the shape of the input is (8, 1, 28, 28).

Let's look at some of the training images.

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

# get some random training images
dataiter = iter(mnist_dataloader)
for images, labels in mnist_dataloader:
    break
# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(8)))
```



5 0 4 1 9 2 1 3

5.1.2 2.2 Defining the Neural Network

PyTorch provides a `nn.Module` that builds neural networks. Now, we will use it to define our network class.

[ ]:
```
import torch.nn as nn


class Net(nn.Module):
    def __init__(self, activation=nn.Sigmoid(),
                 input_size=1*28*28, hidden_size=100, classes=10):

        super().__init__()
        self.input_size = input_size

        # Here we initialize our activation and set up our two linear layers
        self.activation = nn.Sigmoid() # sigmoid function
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, classes)

    def forward(self, x):
        x = x.view(-1, self.input_size) # flatten the images into a 1D array␣
↪(tensor), 28x28 = 784
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)

        return x
```

Looking at the constructor of `Net`, we have, - `super().__init__` creates a class that inherits attributes and behaviors from another class.

- `self.fc1` creates an affine layer with `input_size` inputs and `hidden_size` outputs.
- `self.fc2` is the second affine layer.

The `Forward` function defines the forward pass of the mode.:

- Input `x` is flattened with `x = x.view(-1, self.input_size)` to be able to use as input to the affine layer.
- Apply `fc1`, `activation`, `fc2` sequentially to complete the network.

Central to all neural networks in PyTorch is the `autograd` package. It provides automatic differentiation for all operations on Tensors. If we set the attribute `.requires_grad` of `torch.Tensor` as `True`, it tracks all operations applied on that tensor. Once all the computations are finished, the function `.backward()` computes the gradients into the `Tensor.grad` variable

Thanks to the autograd package, we just have to define the forward() function. We can use any of the Tensor operations in the forward() function. The backward() function (where gradients are computed through back-propagation) is automatically defined by PyTorch.

We can use `print()` to look at all the defined layers of the network (but it won't show the information of the forward pass).

The learned parameters of a model are returned by `[model_name].parameters()`. We can also access the parameters of different layers by `[model_name].[layer_name].parameters()`.

Let's create an instance of the `Net` model and look at the parameters matrix shape for each of the layers.

[ ]:
```
net = Net()
# Always remember to move the network to the GPU/CPU depending on device
net = net.to(device)

print(net)

print("Shapes of the Parameter Matrix:")
for parameter in net.parameters():
        print(parameter.shape)
```

```
Net(
  (activation): Sigmoid()
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
)
Shapes of the Parameter Matrix:
torch.Size([100, 784])
torch.Size([100])
torch.Size([10, 100])
torch.Size([10])
```

5.2 2.3 Defining the Loss function and optimizer

Since it is a multi-class classification, we will use the Cross-Entropy loss and optimize it using SGD with momentum. You don't need to understand the details of the loss function and optimizer for now, but if you are interested, you can refer to the documentation for the loss function and documentation for the optimizer. Suffice it to say that the loss function measures the error between

the predicted labels and the actual labels, and the optimizer updates the weights of the network to minimize this error.

The `torch.nn` and `torch.optim` modules include a variety of loss functions and optimizers. We will initialize an instance of them.

[ ]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

5.3 2.4 Training the network

We have completed setting up the dataloader, loss function as well as the optimizer. We are now all set for training the network.

[ ]:
```
# Initializing the list for storing the loss and accuracy

train_loss_history = [] # loss
train_acc_history = [] # accuracy

for epoch in range(2):


    running_loss = 0.0
    correct = 0.0
    total = 0

    # Iterating through the minibatches of the data

    for i, data in enumerate(mnist_dataloader, 0):

        # data is a tuple of (inputs, labels)
        X, y = data

        X = X.to(device)
        y = y.to(device)

        # Reset the parameter gradients  for the current  minibatch iteration
        optimizer.zero_grad()


        y_pred = net(X)             # Perform a forward pass on the network␣
↪with inputs
        loss = criterion(y_pred, y) # calculate the loss with the network␣
↪predictions and ground Truth
        loss.backward()             # Perform a backward pass to calculate the␣
↪gradients
```

```
        optimizer.step()            # Optimize the network parameters with␣
↪calculated gradients


        # Accumulate the loss and calculate the accuracy of predictions
        running_loss += loss.item()
        _, preds = torch.max(y_pred, 1) #convert output probabilities of each␣
↪class to a singular class prediction
        correct += preds.eq(y).sum().item()
        total += y.size(0)

        # Print statistics to console
        if i % 1000 == 999: # print every 1000 mini-batches
            running_loss /= 1000
            correct /= total
            print("[Epoch %d, Iteration %5d] loss: %.3f acc: %.2f %%" %␣
↪(epoch+1, i+1, running_loss, 100*correct))
            train_loss_history.append(running_loss)
            train_acc_history.append(correct)
            running_loss = 0.0
            correct = 0.0
            total = 0

print('FINISH.')
```

```
[Epoch 1, Iteration  1000] loss: 1.749 acc: 57.64 %
[Epoch 1, Iteration  2000] loss: 0.910 acc: 79.27 %
[Epoch 1, Iteration  3000] loss: 0.612 acc: 85.62 %
[Epoch 1, Iteration  4000] loss: 0.527 acc: 86.85 %
[Epoch 1, Iteration  5000] loss: 0.447 acc: 88.31 %
[Epoch 1, Iteration  6000] loss: 0.421 acc: 88.42 %
[Epoch 1, Iteration  7000] loss: 0.389 acc: 89.21 %
[Epoch 2, Iteration  1000] loss: 0.335 acc: 90.50 %
[Epoch 2, Iteration  2000] loss: 0.370 acc: 89.34 %
[Epoch 2, Iteration  3000] loss: 0.321 acc: 90.81 %
[Epoch 2, Iteration  4000] loss: 0.344 acc: 90.05 %
[Epoch 2, Iteration  5000] loss: 0.315 acc: 90.95 %
[Epoch 2, Iteration  6000] loss: 0.322 acc: 90.66 %
[Epoch 2, Iteration  7000] loss: 0.309 acc: 91.21 %
FINISH.
```

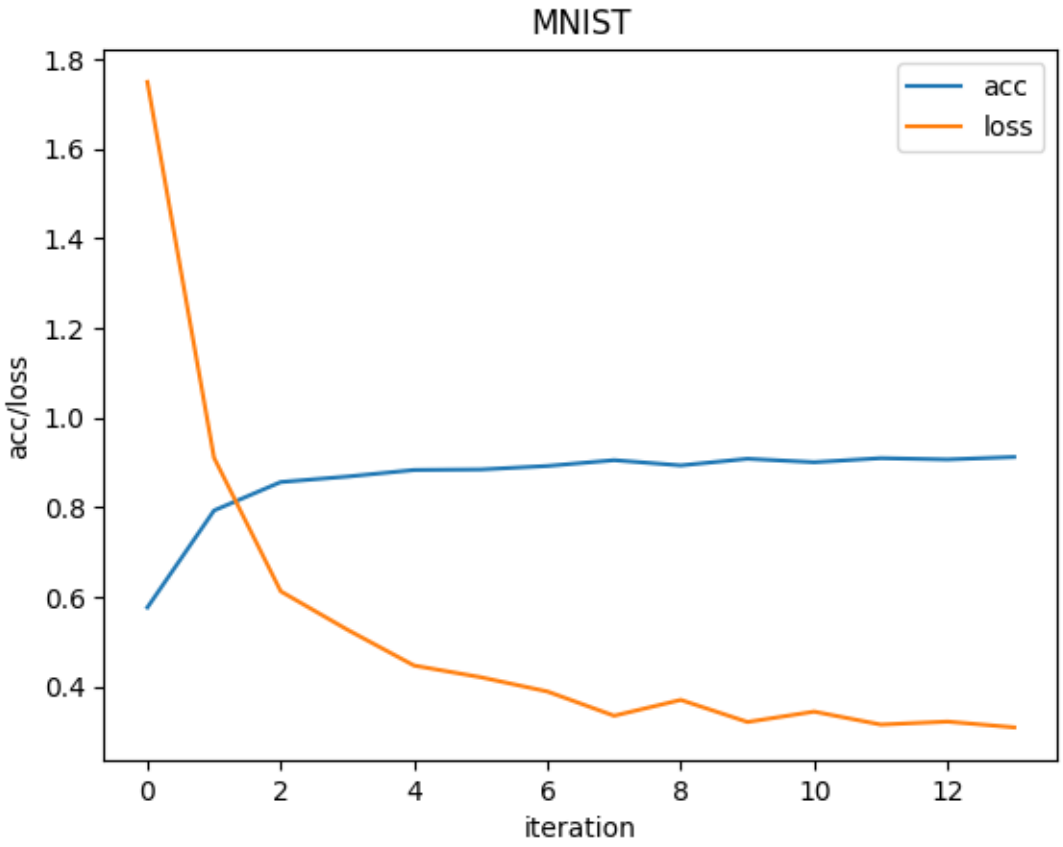
So the general training pass is summarized below:

- `zero_grad()`: Zero the gradient buffers of all the model parameters to start the current minibatch iteration.
- `y_pred = net(X)`: Make a forward pass through the network by passing the images to the model to get the predictions, which are log probabilities of image belonging to each of the class.

- `loss = criterion(y_pred, y)`: Calculate the loss from the generated predictions and the training data `y`.
- `loss.backward()`: Perform a backward pass through the network to calculate the gradients for model parameters.
- `optimizer.step()`: Do an optimization step to update the model parameters using the calculated gradients.

We keep tracking the training loss and accuracy over time. The following plot shows average values for train loss and accuracy.

[ ]:
```
plt.plot(train_acc_history)
plt.plot(train_loss_history)
plt.title("MNIST")
plt.xlabel('iteration')
plt.ylabel('acc/loss')
plt.legend(['acc', 'loss'])
plt.show()
```



5.4 2.5 Testing the performance of the model

We have trained the network for 2 passes over the entire training dataset. Let's check the model performance using the test data. We will pass the test data to the model to predict the class label and check it against the ground-truth.

[ ]:
```
# obtain one batch of test images
dataiter = iter(mnist_test_dataloader)
images, labels = dataiter.__next__()
images, labels = images.to(device), labels.to(device)

# get sample outputs
outputs = net(images)
# convert output probabilites to predicted class
_, predicted = torch.max(outputs, 1)
```

We will visualize the results to display the test images and their labels in the following format: `predicted (ground-truth)`. The text will be green for accurately classified examples and red for incorrect predictions.

[ ]:
```
# prep images for display
if not isinstance(images, np.ndarray):
    images = images.cpu().numpy()

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25,4))
for idx in range(8):
    ax = fig.add_subplot(2, 8//2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title(f"{classes[predicted[idx]]} ({classes[labels[idx]]})",
                 color="green" if predicted[idx]==labels[idx] else "red")
```

Let's find which classes of images performed well, and the classes that did not perform well! `torch.no_grad()` makes sure that gradients are not calculated for the tensors since we only are performing a forward pass.

[ ]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
```

```
with torch.no_grad():
    for data in mnist_test_dataloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(8):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %11s: %2d %%' % (classes[i], 100 * class_correct[i] /␣
 ↪class_total[i]))
```

```
Accuracy of           0: 98 %
Accuracy of           1: 98 %
Accuracy of           2: 88 %
Accuracy of           3: 91 %
Accuracy of           4: 91 %
Accuracy of           5: 86 %
Accuracy of           6: 94 %
Accuracy of           7: 88 %
Accuracy of           8: 83 %
Accuracy of           9: 92 %
```

That's the end of the `PyTorch` Tutorial! Now you have a basic understanding of PyTorch and how to train your own neural networks using PyTorch.

5.5 References

1. PyTorch Tutorial
2. MNIST dataset training using PyTorch