Report Title: **The Introduction of Pytorch**

Author Name: **王小慧**

Student ID: **21307110415**

Institute: **信息科学与工程学院**

Report Date: **2024/11/11**

Contents

1 Introduction

PyTorch is an open-source machine learning library widely used in research and industry due to its superiority of flexibility, and GPU acceleration. This report discusses the results of using PyTorch to conduct a series of experiments. Our primary objectives are to familiarize ourselves with basic PyTorch operations, including tensor manipulations, data transformations, and neural network construction. This setup and experiment aimed to explore how PyTorch facilitates deep learning workflows and efficient model training.

2 Experimental Setup

The experiments were conducted on two different operating systems: Windows and Linux. Both systems used Anaconda to create environments. The Fashion-MNIST dataset was selected as the primary dataset. Table 1 contains the specific details of the setup:

Table 1 Setups for Windows and Linux

Windows	Linux
Python Version: 3.8	Python Version: 3.9
Torch Version: 1.13.1+cu116	Torch Version: 2.1.0+cu12
GPU: RTX 3050ti	GPU: RTX 3090

Other libraries that were used in experiments:

- Torchvision: To download and preprocess datasets.
- Numpy: For data manipulation to facilitate tensor operations.
- Matplotlib: For visualization.

3 Experimental Procedure

3.1 Installation of Torch

We first installed Torch and its dependencies and verified the installation using the code below:

```
1 import torch
2 print(torch.cuda.is_available)
```

3.2 Tensor Initialization and Manipulation

We explored PyTorch' s tensor structure, which is fundamental to PyTorch operations. Several tensors were created from different sources, including lists and NumPy arrays, to understand tensor attributes like shape, dtype, and device.

Initialization and Manipulation

```
'''Initialization'''
# Initialize tensor from list
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)

# Initialize tensor from numpy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)

# Initialize tensor from another tensor
x_ones = torch.ones_like(x_data)
x_rand = torch.rand_like(x_data, dtype=torch.float32)

'''Manipulation'''
# Attributes
print(f"Shape of x_data: {x_data.shape}")
print(f"Datatype of x_data: {x_data.dtype}")
print(f"Device x_data is stored on: {x_data.device}")

# Slice
print(f"First row: {x_data[0]}")
print(f"First column: {x_data[:, 0]}")
print(f"Last column: {x_data[:, -1]}")

# Concat
a = torch.ones(3,2)
b = torch.ones(3,2)
c = torch.ones(3,2)
d = torch.cat([a,b,c], dim=0)
e = torch.cat([a,b,c], dim=1)
print(f"Shape of d: {d.shape}\n Shape of e: {e.shape}")

# Matrix Multiplication
a[0][1] = 3
a[2][1] = 2
b[1][0] = 5
b[2][1] = 4
print("a:\n", a, "\nb:\n", b)
print("The result of a * b:\n", a * b)  # element-wise multiplication
print("The result of a @ b:\n", a @ b.T)  # matrix multiplication
print("The result of a.matmul(b):\n", a.matmul(b.T))  # matrix multiplication
```

Output

```
Shape of x_data: torch.Size([2, 2])
Datatype of x_data: torch.int64
Device x_data is stored on: cpu

First row: tensor([1, 2])
First column: tensor([1, 3])
Last column: tensor([2, 4])

Shape of d: torch.Size([9, 2])
Shape of e: torch.Size([3, 6])

a:
 tensor([[1., 3.],
        [1., 1.],
        [1., 2.]])
b:
 tensor([[1., 1.],
        [5., 1.],
        [1., 4.]])

The result of a * b:
 tensor([[1., 3.],
        [5., 1.],
        [1., 8.]])
The result of a @ b:
 tensor([[ 4.,  8., 13.],
        [ 2.,  6.,  5.],
        [ 3.,  7.,  9.]])
The result of a.matmul(b):
 tensor([[ 4.,  8., 13.],
        [ 2.,  6.,  5.],
        [ 3.,  7.,  9.]])
```

3.3 Data Loading and Transformation

Using the torchvision library, we loaded the Fashion-MNIST dataset. We applied standard transformations, including normalization and data augmentation. The data loader was configured to provide data in batches for training and testing.

Data Loading and Transformation

```
train_transform = transforms.Compose([
    transforms.RandomRotation(10),  # randomly rotate
    transforms.RandomHorizontalFlip(),  # randomly flip
    transforms.ToTensor(),  # image to tensor
    transforms.Normalize((0.5,), (0.5,))  # mean->0.5, std->0.5
])
```

```
    test_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])

    train_data = datasets.FashionMNIST(root=args.data_path, train=True, download=
        True, transform=train_transform)
    test_data = datasets.FashionMNIST(root=args.data_path, train=False, download=
        True, transform=test_transform)

    train_dataloader = DataLoader(train_data, batch_size=args.batch_size, shuffle=
        True)
    test_dataloader = DataLoader(test_data, batch_size=args.batch_size, shuffle=
        False)
```

3.4 Model Construction

We constructed a MLP network using the torch.nn module, consisting of several fully connected layers and ReLU activations. This model was designed to classify the Fashion-MNIST images into ten categories.

Model Construction

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten() # Needed in Linear Layers
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

We can use other models to classify Fashion-MNIST dataset, but we will leave this exploration to future experiment.

3.5 Model Training and Evaluation

We implemented a training loop to iterate over multiple epochs, using the autograd feature for back propagation. Following training, the model was evaluated on the test set to measure accuracy and loss.

Training and Testing Loop

```python
def train_loop(dataloader, model, loss_func, optimizer, device, batch_size, epoch_losses):
    size = len(dataloader.dataset)
    model.train()
    epoch_loss = 0
    for batch, (x,y) in enumerate(dataloader):
        x, y = x.to(device), y.to(device)
        pred = model(x)
        loss = loss_func(pred, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        epoch_loss += loss.item()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(x)
            print(f"loss:{loss:>7} [{current:>5d} / {size:>5d}]")

    epoch_losses.append(epoch_loss / len(dataloader))

def test_loop(dataloader, model, loss_func, device, test_losses, accuracies):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for x, y in dataloader:
            x, y = x.to(device), y.to(device)
            pred = model(x)
            test_loss += loss_func(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test: \n Accuracy:{(100*correct):>0.1f}%, Avg loss:{test_loss:>8f} \n")

    test_losses.append(test_loss)
    accuracies.append(correct)
```

4 Results and Analysis

4.1 Training Details

Model Architecture and Optimizer:

- Model: MLP (4x Linear+ReLU and an output layer)
- Loss Function: Cross-Entropy
- Optimizer: Adam

Hyper Parameters:

- Learning Rate = 0.003
- Batch Size = 64
- Epochs = 10

4.2 Results

The curves depicting the training loss, test loss, and test accuracy are shown in Figure 1.

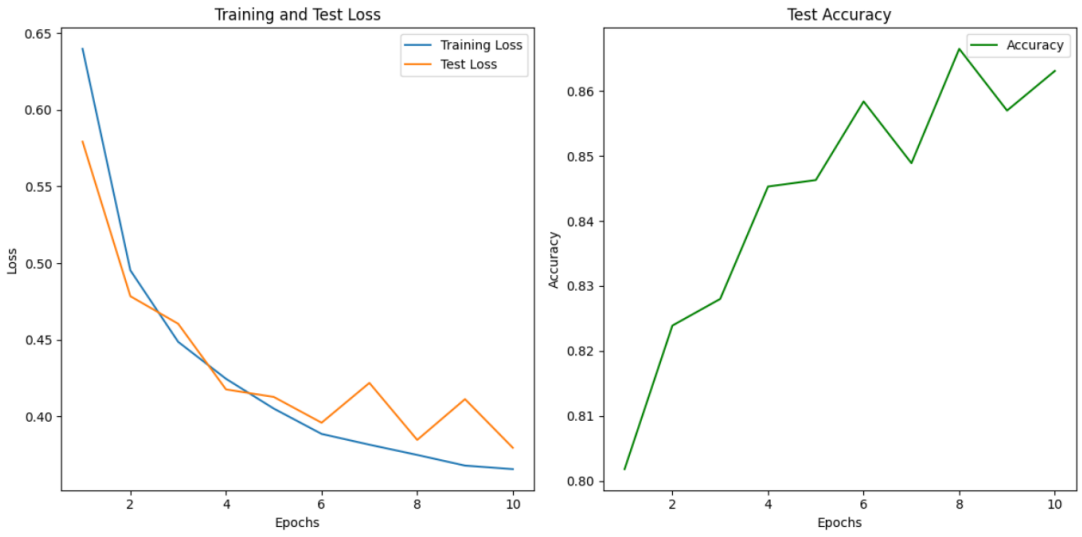

Figure 1 : illustrates the evolution of the loss and accuracy throughout the 10 epochs. As the number of epochs increases, the training loss steadily decreases, indicating that the model is learning to minimize its prediction error. Concurrently, the test accuracy improves, demonstrating that the model is successfully generalizing to unseen data. This trend reflects effective model convergence and learning over time.

4.3 Analysis

The model achieved an accuracy of approximately 86% on the Fashion-MNIST test set. Analysis of the training curves indicates that the model' s performance improved with each epoch, suggesting that the neural network effectively learned to classify the images. Loss values consistently decreased, showing that the network were successfully optimized without over-fitting. (It is normal to have some minor fluctuations.) Additionally, the model showed potential for further improvement with additional tuning of hyper parameters, such as learning rate and batch size, or by modifying the model architecture, we will explore it in Experiment 2.

5 Conclusion

This experiment provided hands-on experience with PyTorch, from tensor manipulations to constructing and training a neural network model. PyTorch' s dynamic computation graph and ease of tensor operations simplified the training process, demonstrating its strength in deep learning applications. While the model achieved satisfactory results, future experiments could explore more complex architectures or techniques like CNN, ResNet.