Convolutional Neural Networks

Part II

CNN in Pytorch

Data

The Fashion MNIST dataset is a collection of images that includes various clothing items, commonly used for benchmarking machine learning models.

It consists of 70k grayscale images (60k + 10k), each of size 28x28 pixels, composed of pixels with values ranging from 0 to 255 in a Numpy array format.

The labels are integers from 0 to 9, representing the following clothing items:

0: T-shirt/top

1: Trouser

2: Pullover

3: Dress

4: Coat

5: Sandal

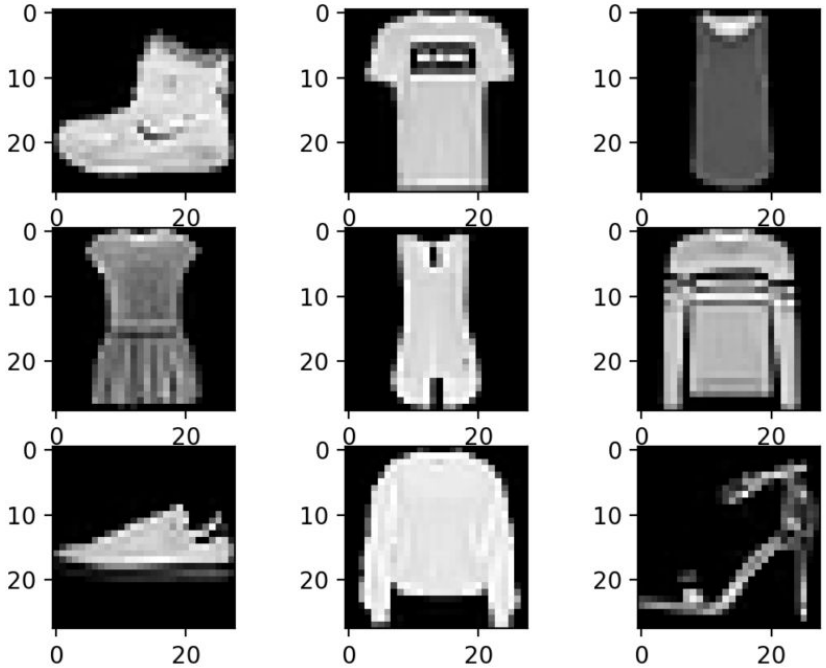
6: Shirt

7: Sneaker

8: Bag

9: Ankle boot

Fashion MNIST



Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms # for preprocessingg
from torch.utils.data import Dataset, DataLoader # gives easier dataset managment
```

CPU vs GPU

Using a GPU is recommended for training machine learning models on datasets like Fashion MNIST because GPUs can process data much faster than CPUs.

This is due to their ability to perform multiple calculations simultaneously, which significantly speeds up the training process for complex models and large datasets.

GPU acceleration is supported by major deep learning frameworks, making it an efficient choice for faster model development and experimentation.

GPU

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

Multi-GPU

When using a single GPU, the following code is typically used:

```
[ ] device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = Net()
    model.to(device)
```

However, when using multiple GPUs, you would use nn.DataParallel like this:

```
[ ] device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = Net()
    if torch.cuda.device_count() > 1:
        model = nn.DataParallel(model)
    model.to(device)
```

When using nn.DataParallel, the batch size is automatically distributed across the GPUs. Therefore, it's advisable to increase the batch size proportionally to the number of GPUs.

Data Download

```
train_dataset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())

test_dataset = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())
```

Set the download location: root='./data' sets the directory where the FashionMNIST dataset will be downloaded and stored. If the directory does not exist, it will be created.

Specify the dataset split: The train=True or train=False parameter determines whether the training dataset or the test dataset is being downloaded. train=True downloads the training set, while train=False downloads the test set.

Download the dataset: download=True checks if the dataset already exists at the specified location (root). If it does not, the dataset will be downloaded from the internet. If the dataset is already present, it will not be downloaded again.

Transform the images: transform=transforms.ToTensor() converts the images in the dataset to PyTorch tensors. This transformation converts the image pixel values from a range of 0-255 to a float tensor with a range of 0.0 to 1.0. This is a common preprocessing step for neural network training.

DataLoader

```
batch_size = 100

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

torch.utils.data.DataLoader

Set the batch size: `batch_size = 100` sets the number of samples that will be passed through the network at one time. It is a critical parameter that can affect model training efficiency and effectiveness. A batch size of 100 means that 100 images will be used to compute the gradient and update the model parameters during each iteration of the training loop.

Create the training data loader:

```
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
```

The DataLoader is a PyTorch utility that abstracts the complexity of handling datasets in batches. Here, it is configured to use the `train_dataset` loaded earlier, with the specified `batch_size`. The `shuffle=True` parameter ensures that the data is shuffled at the beginning of each training epoch, which helps to reduce overfitting and improve model generalization.

torch.utils.data.DataLoader

Create the testing data loader:

```
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

Similarly, this line creates a data loader for the test dataset. The key difference is `shuffle=False`, which is typical for testing and validation data loaders since shuffling has no benefit when evaluating the model.

Simple Neural Network Model without nn.Conv2d

```
class FashionDNN(nn.Module):
    def __init__(self):
        super(FashionDNN, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Simple Neural Network Model without nn.Conv2d

Class Definition:

`class FashionDNN(nn.Module)` defines a new class called `FashionDNN` that inherits from `nn.Module`, the base class for all neural network modules in PyTorch.

This inheritance provides useful methods and attributes for your network class.

Simple Neural Network Model without nn.Conv2d

Constructor `__init__(self)`:

`super(FashionDNN, self).__init__()` calls the constructor of the parent class (nn.Module), allowing the class to inherit all its functionalities.

`self.fc1 = nn.Linear(784, 256)` defines the first fully connected (linear) layer with 784 input features and 256 output features. The input size of 784 corresponds to the flattened size of the 28x28 images from the FashionMNIST dataset.

`self.fc2 = nn.Linear(256, 128)` defines the second fully connected layer with 256 inputs (the outputs of fc1) and 128 outputs.

`self.fc3 = nn.Linear(128, 10)` defines the third and final fully connected layer with 128 inputs (the outputs of fc2) and 10 outputs, corresponding to the 10 classes of the FashionMNIST dataset.

Simple Neural Network Model without nn.Conv2d

Forward Pass `forward(self, x)`:

The forward method defines how the input `x` flows through the network.

`x = x.view(-1, 784)` flattens the input tensor to shape (-1, 784), where -1 infers the correct batch size and 784 is the flattened image size.

`x = F.relu(self.fc1(x))` passes the flattened input through the first linear layer (fc1) and then applies the ReLU (Rectified Linear Unit) activation function to the output.

`x = F.relu(self.fc2(x))` further processes the output of the first layer through the second linear layer (fc2) followed by another ReLU activation.

`x = self.fc3(x)` passes the output of the second layer through the final linear layer (fc3), producing the final output of the network. This output can then be passed to a loss function for training or an activation function like softmax for inference to get probabilities for each class.

Defining Parameters

1. Loss function
2. Learning rate
3. Optimizer

```
learning_rate = 0.001
model = FashionDNN()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
print(model)
```

```
cuda:0
FashionDNN(
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=10, bias=True)
)
```

Training

```
num_epochs = 10

train_acc = []
loss_list = []

for epoch in range(num_epochs):
    losses = 0
    correct = 0
    for images, labels in train_loader:
        train, labels = images.to(device), labels.to(device)
        outputs = model(train)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == labels).sum().item()

    train_acc.append(correct / len(train_dataset))
    loss_list.append(losses / (len(train_dataset)/batch_size))
```

Training

The outer loop `for epoch in range(num_epochs):` iterates over each epoch.

Inside this loop, `losses = 0` and `correct = 0` are initialized to accumulate the total loss and the number of correctly classified samples within the epoch.

The inner loop `for images, labels in train_loader:` iterates over each batch of images and labels from the training dataset.

`train, labels = images.to(device), labels.to(device)` moves the images and labels to the designated computing device (GPU or CPU), as specified by the device variable earlier in the code.

`outputs = model(train)` computes the forward pass through the model, getting the predicted outputs for the input images.

Training

`loss = criterion(outputs, labels)` calculates the loss between the predicted outputs and the actual labels using a predefined loss function criterion.

`optimizer.zero_grad()` clears old gradients; without this, gradients would accumulate across batches.

`loss.backward()` computes the gradient of the loss with respect to the model parameters.

`optimizer.step()` updates the model parameters based on the gradients.

`losses += loss.item()` accumulates the total loss for the epoch.

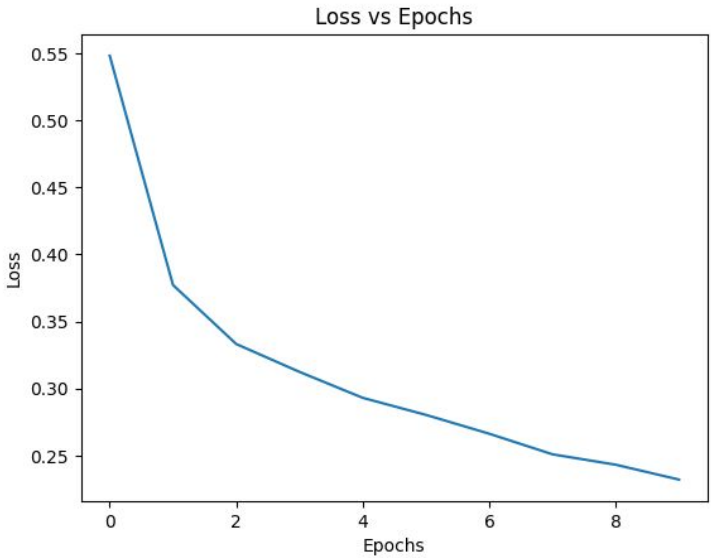
`_, predicted = torch.max(outputs.data, 1)` finds the predictions with the highest score for each input image.

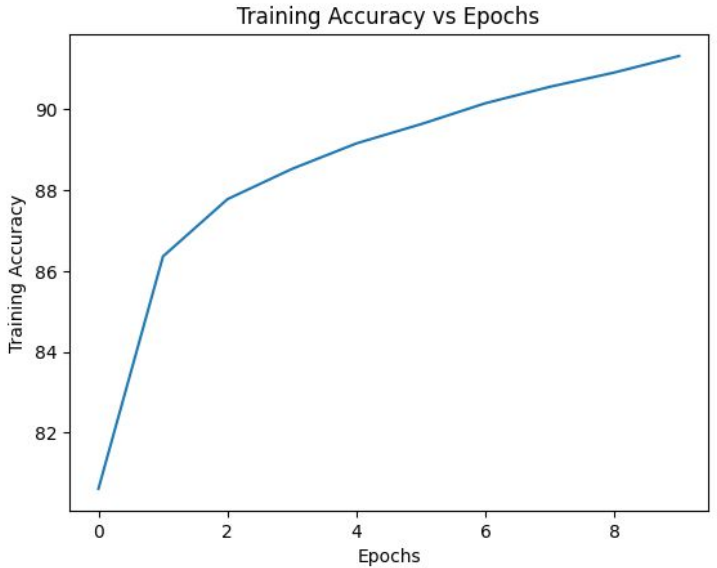
`correct += (predicted == labels).sum().item()` calculates the number of correct predictions in the batch and accumulates it over the epoch.

Results

```
plt.plot(loss_list)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss vs Epochs')
plt.show()
plt.plot(train_acc)
plt.xlabel('Epochs')
plt.ylabel('Training Accuracy')
plt.title('Training Accuracy vs Epochs')
plt.show()
```

Results


Loss vs Epochs
Loss
Epochs
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0
2
4
6
8


Training Accuracy vs Epochs
Training Accuracy
Epochs
90
88
86
84
82
0
2
4
6
8

Convolutional Neural Network Model

```
class FashionCNN(nn.Module):
    def __init__(self):
        super(FashionCNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer3 = nn.Sequential(
            nn.Linear(in_features=64*6*6, out_features=600),
            nn.Linear(600, 120),
            nn.Linear(120, 10)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.layer3(x)
        return x
```

CNN Model

`super(FashionCNN, self).__init__()` initializes the parent class (nn.Module), allowing the class to inherit functionalities from PyTorch's module class.

Layer 1: A sequential container that includes:

A convolutional layer (`nn.Conv2d`) with 1 input channel (grayscale image), 32 output channels, a kernel size of 3, and padding of 1. This layer is designed to extract low-level features from the image.

Batch normalization (`nn.BatchNorm2d`) with 32 features to stabilize learning by normalizing the input to the ReLU activation function.

A ReLU activation function (`nn.ReLU()`) to introduce non-linearity into the model, allowing it to learn complex patterns.

A max pooling layer (`nn.MaxPool2d`) with a kernel size of 2 and stride of 2, reducing the spatial dimensions by half and helping to make the representation size manageable.

CNN Model

Layer 2: Another sequential container that includes:

A convolutional layer with 32 input channels, 64 output channels, a kernel size of 3, and no padding specified. This increases the depth of the feature maps while capturing more detailed features.

Batch normalization with 64 features + A ReLU activation function.

Another max pooling layer, further reducing the spatial dimensions of the feature maps.

Layer 3: A sequential container for the fully connected layers, transitioning from feature extraction to classification:

A linear layer (`nn.Linear`) that flattens the output of the last max pooling layer (*assuming the input image size leads to a feature map size of 6x6 after two pooling layers*) to 600 features.

Another linear layer reducing the feature dimension from 600 to 120.

A final linear layer that outputs 10 features, corresponding to the 10 classes of the FashionMNIST dataset.

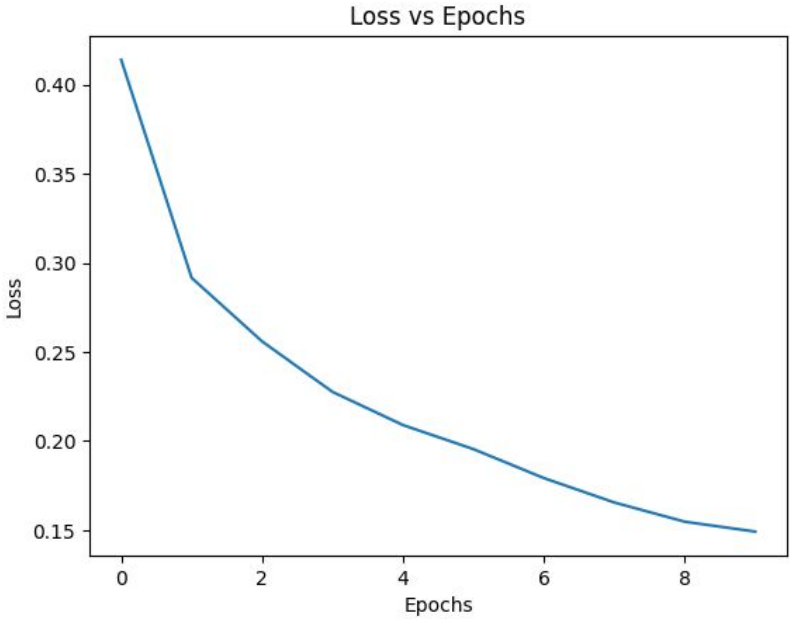
CNN Model and Parameter Settings

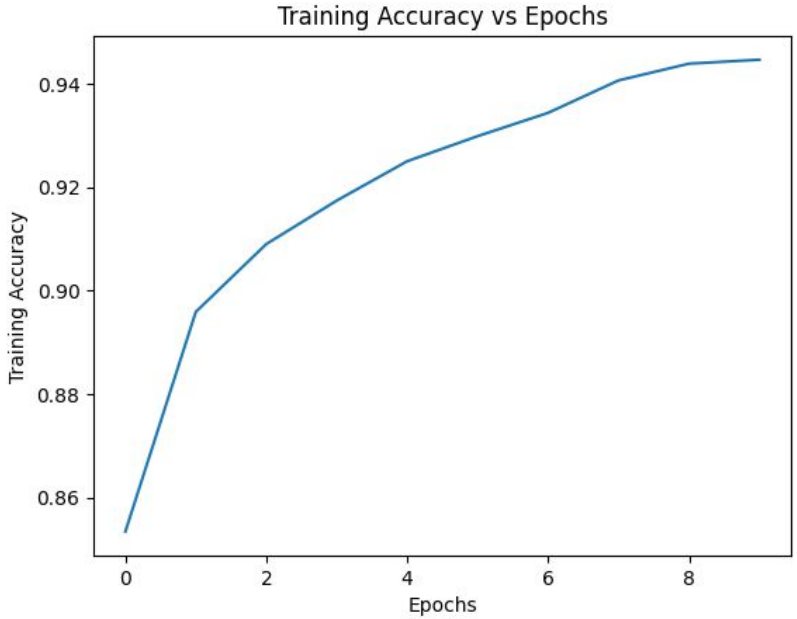
```
learning_rate = 0.001
model = FashionCNN()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

from torchsummary import summary
summary(model, (1, 28, 28))
```

```
cuda:0
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
       BatchNorm2d-2           [-1, 32, 28, 28]              64
              ReLU-3           [-1, 32, 28, 28]               0
         MaxPool2d-4           [-1, 32, 14, 14]               0
            Conv2d-5           [-1, 64, 12, 12]          18,496
       BatchNorm2d-6           [-1, 64, 12, 12]             128
              ReLU-7           [-1, 64, 12, 12]               0
         MaxPool2d-8             [-1, 64, 6, 6]               0
            Linear-9                  [-1, 600]       1,383,000
           Linear-10                  [-1, 120]          72,120
           Linear-11                   [-1, 10]           1,210
================================================================
Total params: 1,475,338
Trainable params: 1,475,338
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.86
Params size (MB): 5.63
Estimated Total Size (MB): 6.49
----------------------------------------------------------------
```

Results


Loss vs Epochs
Loss
0.40
0.35
0.30
0.25
0.20
0.15
0
2
4
6
8
Epochs


Training Accuracy vs Epochs
Training Accuracy
0.94
0.92
0.90
0.88
0.86
0
2
4
6
8
Epochs