澳門城市大學
Universidade da Cidade de Macau
City University of Macau

Task 2 Report

Faculty:	Faculty of Data Science
Major:	Computer Science
Name:	Yuchen Shi

Friday 13th September, 2024

Contents

1 PyTorch

1.1 Introduction – What is PyTorch?

Figure 1: PyTorch

PyTorch is defined as an open source machine learning library for Python. It is designed to provide maximum flexibility and speed during the development of machine learning models.

1.2 Why PyTorch?

- **Dynamic Computational Graphs:** PyTorch uses dynamic computational graphs, which means that the graph is created on the fly. This allows for more flexibility and ease of use.
- **Strong GPU Support:** PyTorch has strong support for GPUs, which allows for faster training of models. We can use CUDA to speed up the training process and CUDNN to speed up the convolutional neural networks.
- **Native Optimization libraries:** PyTorch has native optimization libraries that allow for faster training of models. PyTorch is its robust native optimisation library, which provides efficient and well-tuned implementations of numerous popular optimisation algorithms. This means you can easily experiment with different optimisation methods, such as stochastic gradient descent, Adam, or RMSProp, to find the best one for your machine learning model.

Figure 2: NVIDIA

2 PyToch's basic functions

2.1 Tensor

2.1.1 What is Tensor?

Tensor describes a multilinear relationship between sets of algebraic objects related to a vector space.

2.1.2 Tensor in PyTorch

Tensor is a soul of PyTorch. It's like to Numpy Arrays. Tensors are created on CPU but in PyTorch we can move them to GPU for faster computations.

```
import torch
if torch.cuda.is_available():
tensor = tensor.to('cuda')
```

The above code can move the tensor to the GPU if available.

2.1.3 Initializing Tensor

- **Directly from data:** Tensors can be created directly from data. The data type is automatically inferred

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

- **From a NumPy array:** Tensors created by Numpy arrays.

```
import numpy as np
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

- **From another tensor:** The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

```
x_ones = torch.ones_like(x_data)
x_rand = torch.rand_like(x_data, dtype=torch.float)
```

- **With random or constant values:** shape is a tuple of tensor dimensions. In the functions below, it determines the dimensionality of the output tensor.

```
shape = (2,3,)# 2 rows, 3 columns
rand_tensor = torch.rand(shape)# random numbers
ones_tensor = torch.ones(shape)# ones
zeros_tensor = torch.zeros(shape)# zeros
```

2.1.4 Operations on Tensors

- **indexing and slicing:** Tensors can be indexed and sliced just like Numpy API.

```
tensor = torch.ones(4, 4) # 4x4 matrix of ones
print(f"First row: {tensor[0]}") # First row
print(f"First column: {tensor[:, 0]}") # First column
print(f"Last column: {tensor[..., -1]}")  # Last column
tensor[:,1] = 0 # change the second column to zeros
print(tensor)
```

- **Joining tensors:** Use torch.cat to concatenate a sequence of tensors along a given dimension.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

- **Arithmetic operations:** PyTorch tensors support all the standard arithmetic operations.

```
y1 = tensor @ tensor.T # this return the transpose of
                                the tensor
y2 = tensor.matmul(tensor.T)
y3 = torch.rand_like(tensor)
torch.add(tensor, tensor, out=y3)
```

- **In-place operations:** Operation that store the result into the operand.

```
tensor.add_(5)
```

2.1.5 Bridge with NumPy

Tensor and Numpy can share thier memory locations. ANd changing one will change the other.

- **Tensor to NumPy array**

```
t = torch.ones(5)
n = t.numpy()
```

- **NumPy array to Tensor**

```
n = np.ones(5)
t = torch.from_numpy(n)
```

2.2 Datasets and Dataloaders

```
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

2.2.1 Datasets

```
torch.utils.Dataset
```

In a dataset, we have four basic paraments:

- **root:** the path where the data is stored
- **tranin:** specifies training or test dataset
- **download:** if the root is empty then download the dataset form the internet
- **transform and target-transform:** specify the feature and label transformations

We use matplotlib to visualize the samples in our training data

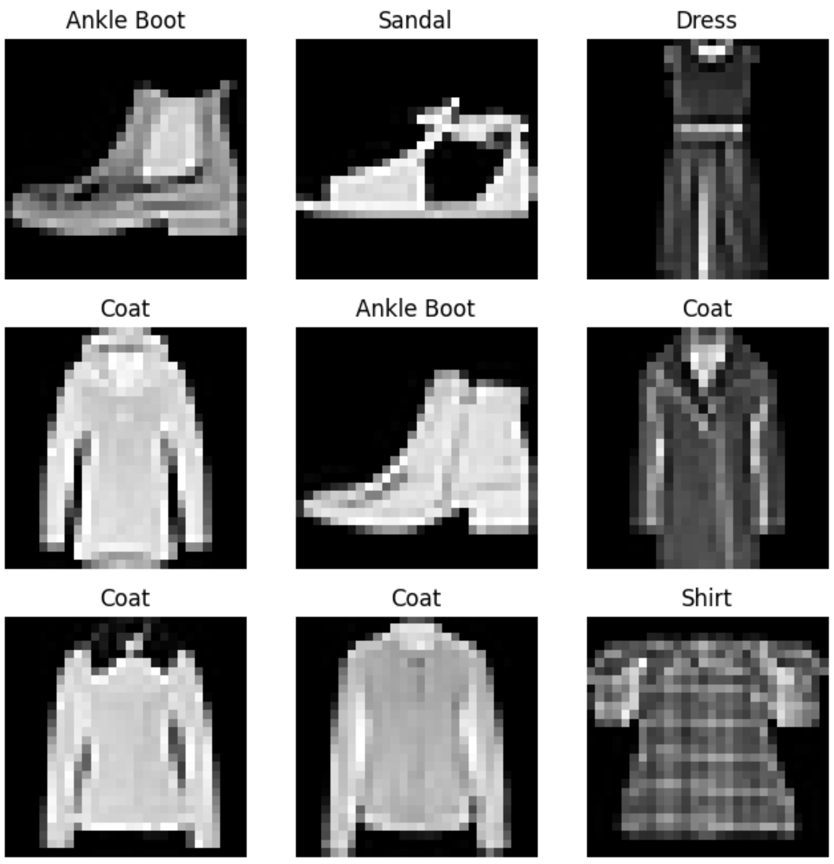


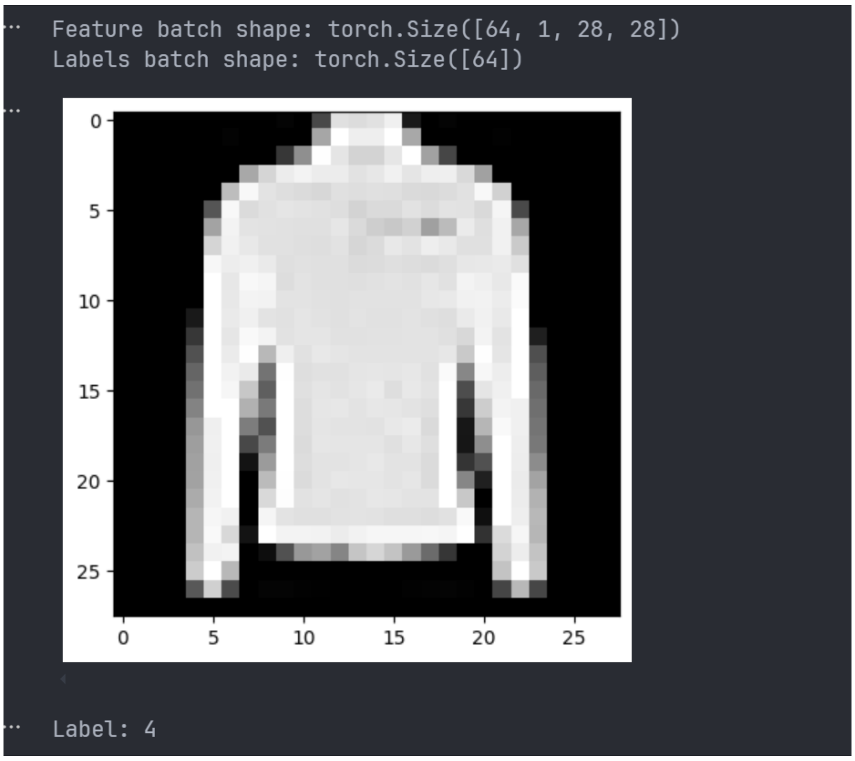
Figure 3: visualize the samples by matplotlib

2.2.2 Dataloaders

Dataloader is an iterable that abstracts the process of batching, shuffling, and loading the data. The dataset retrieves the dataset's features and labels one sample at a time. When we are training the model, we want reshuffle the data at every epoch to reduce model overfitting.

```
from torch.utils.data import DataLoader
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=
                                       True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

When we loaded the data into the dataloader, we specify the batch size and shuffle. Each iteration of the dataloader will return a batch of train features and labels.



```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
```

```
Label: 4
```

Figure 4: Dataloader

2.3 Transforms

We always need to convert the data into tensors. When we load the data, we can use the transforms to convert the data into tensors. And we can also normalize and standardize the data.

2.3.1 How to use transforms

- **To Tensor:** Converts a PIL image or NumPy ndarray into a FloatTensor. And scales the image's pixel intensity values in the range [0., 1.]

```
    my_transforms = transforms.Compose([transforms.ToTensor()])
```

- **Lambda:** Applies a user-defined lambda function.

```
    target_transform = Lambda(lambda y: torch.zeros(
        10, dtype=torch.float).scatter_(dim=0, index=torch.tensor(y),
                                                value=1))
```

2.4 Building Neural Network

Neural networks are the core of both deep learning and machine learning.

2.4.1 How to build a neural network

- **Perpare the data:** We need to prepare the data before we can use it to train the model.
- **Define a model structure** We define our neural network by subclassing nn.Module, and initialize the neural network layers. Every nn.Module subclass implements the operations on input data in the forward method.
- **Forward propagation:** Calculate outputs through the network.
- **Loss calculation:** Calculate the loss using the model's output and the target. e.g. mean squared error loss (MSE) or cross-entropy loss. We can use the loss function from the torch.nn module. We always hope to minimize the loss.
- **Backward propagation:** Update model parameters to minimize the loss by computing the gradient of the loss. Use optimization algorithms (such as gradient descent, Adam, RMSProp, etc.) for parameter updates.

2.5 Automatic Differentiation

When training neural networks, the most frequently used algorithm is back propagation. In this algorithm, model weights are adjusted according to the gradient of the loss function with respect to the given parameter.

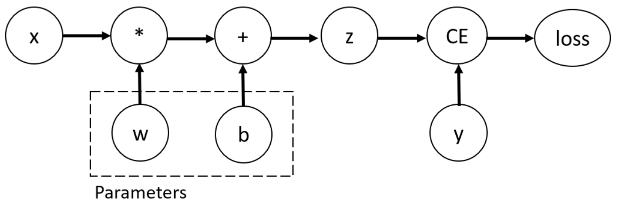


Figure 5: Computational Graph

$$z = x \times w + b \tag{1}$$

$$\text{loss} = CE(z, y) \tag{2}$$

If we in binary classification, CE can be Calculated as:

$$CE = -y \times \log(\hat{y}) - (1 - y) \times \log(1 - \hat{y}) \tag{3}$$

If we in multi-class classification, CE can be Calculated as:

$$CE = -\sum_{i=1}^{n} y_i \times \log(\hat{y}_i) \tag{4}$$

y is binary indicator (0 or 1) if class label i is the correct classification for sample i.

$\hat{y}$ is the predicted probability of class i.

2.5.1 Computing Gradients

We can compute the gradients of the loss with respect to the model parameters by loss.backward(). So that we need $\frac{\partial loss}{\partial w}$ and $\frac{\partial loss}{\partial b}$ under fixed values of x and y.

2.5.2 Disabling Gradient Tracking

By default, all tensors are tracking their computational history and support gradient computation. However, we don't need to track the history for the weights and biases while training the model.

```
with torch.no_grad():
z = torch.matmul(x, w)+b
```

When we want to mark some parameters in your neural network as frozen parameters or speed up computations(only doing forward propagation), we can turn off the gradient computation.

2.6 Optimizers

Training a model is an iterative process; in each iteration the model makes a guess about the output, we use loss function to calculates the error in its guess , collects the derivatives of the error with respect to its parameters, and optimizes these parameters using gradient descent.

2.6.1 Hyperparameters

- **Learning Rate:** how much to update models parameters at each batch/epoch.
- **Batch Size:** the number of data samples propagated through the network before the parameters are updated
- **Epochs:** the number times to iterate over the dataset

2.6.2 Optimization Loop

- **Training loop:** iterate over the training dataset and try to converge to optimal parameters
- **Test loop:** iterate over the test dataset to check if model performance is improving

2.6.3 Loss Function

Loss function measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training.
e.g. Cross-Entropy Loss

```
loss_fn = nn.CrossEntropyLoss()
```

2.6.4 Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. Optimization algorithms define how this process is performed.

In the training loop, optimization happens in the following steps:

- Call optimizer.zero_grad() to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to loss.backward(). PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call optimizer.step() to adjust the parameters by the gradients collected in the backward pass.

2.7 Save and load model

2.7.1 Saving and loaing model weights

PyTorch models store the learned parameters in an internal state dictionary, called state_dict.

```
model = models.vgg16(weights='IMAGENET')
torch.save(model.state_dict(), 'model.pth')
```

To load model weights we use the load_state_dict() method.

```
model = models.vgg166()
model.load_state_dict(torch.load('model.pth'), weights_only=True)
model.eval()
```

2.7.2 Saving and loading the model with shapes

When loading model weights, we needed to instantiate the model class first, because the class defines the structure of a network. We might want to save the structure of this class together with the model, in which case we can pass model to the saving function.

```
torch.save(model, 'model.pth')
```

We can then load the model like this:

```
model = torch.load('model.pth', weights_only=False)
```

Thank You for reading this article.