Deep Learning Framework: PyTorch

DR.FAKHRELDIN SAEED

1

Containt

2

PyTorch

- **PyTorch** is a Python-based machine learning library.
- It consists of two main features:
  - its ability to efficiently perform tensor operations with hardware acceleration (using GPUs)
  - and its ability to build deep neural networks.

3

Datasets & DataLoaders

- ***PyTorch*** provides two data primitives: (that allow you to use pre-loaded datasets as well as your own data)
  - torch.utils.data.DataLoader
  - torch.utils.data.Dataset
- ***Dataset*** stores the samples and their corresponding labels, and ***DataLoader*** wraps an iterable around the Dataset to enable easy access to the samples.
- PyTorch domain libraries provide a number of pre-loaded datasets (such as FashionMNIST) that subclass torch.utils.data.Dataset and implement functions specific to the particular data.
- They can be used to prototype and benchmark our model.
- We can find them here: Image Datasets, Text Datasets, and Audio Datasets

4

Loading a Dataset

- We load the FashionMNIST Dataset with the following parameters:
- ***root*** is the path where the train/test data is stored,
- ***train*** specifies training or test dataset,
- **download=True** downloads the data from the internet if it's not available at root.
- ***transform*** and ***target_transform*** specify the feature and label transformations

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


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```


Creating a Custom Dataset for your files

- A custom Dataset class must implement thre functions:

  __init__, __len__, and __getitem__.
- the FashionMNIST images are stored in a directory img_dir, and their labels are stored separately in a CSV file annotations_file.
- The __init__ function is run once when instantiating the Dataset object. We initialize the directory containing the images, the annotations file, and both transforms.
- The __len__ function returns the number of samples in our dataset.
- The __getitem__ function loads and returns a sample from the dataset at the given index idx

```python
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

TRANSFORMS

- Data does not always come in its final processed form that is required for training machine learning algorithms. We use **transforms** to perform some manipulation of the data and make it suitable for training.

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).scatter_(0,
torch.tensor(y), value=1))
)
```


BUILD THE NEURAL NETWORK

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

- Get Device for Training

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

- Define the Class
- Create an instance of *NeuralNetwork*, and move it to the *device*, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

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

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

AUTOMATIC DIFFERENTIATION WITH *TORCH.AUTOGRAD*

- When training neural networks, the most frequently used algorithm is **back propagation**.
- In this algorithm, parameters (model weights) are adjusted according to the **gradient** of the loss function with respect to the given parameter.
- To compute those gradients, PyTorch has a built-in differentiation engine called ***torch.autograd***.
- It supports automatic computation of gradient for any computational graph.
-

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```


OPTIMIZING MODEL PARAMETERS (1/3)

- **Hyperparameters** are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates.
- hyperparameters for training:
  - ***Number of Epochs*** - the number times to iterate over the dataset
  - ***Batch Size*** - the number of data samples propagated through the network before the parameters are updated
  - ***Learning Rate*** - how much to update models parameters at each batch/epoch.
- Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

OPTIMIZING MODEL PARAMETERS (2/3)

- Once we set our hyperparameters, we can then train and optimize our model with an **optimization loop**.
- Each iteration of the optimization loop is called an **epoch**.
- Each epoch consists of two main parts:
  - The **Train Loop** - iterate over the training dataset and try to converge to optimal parameters.
  - The **Validation/Test Loop** - iterate over the test dataset to check if model performance is improving.
- **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.
- Common loss functions include:
  - nn.MSELoss (Mean Square Error) for regression tasks.
  - nn.NLLLoss (Negative Log Likelihood) for classification.
  - nn.CrossEntropyLoss combines nn.LogSoftmax and nn.NLLLoss.

OPTIMIZING MODEL PARAMETERS (3/3)

- Optimization is the process of adjusting model parameters to reduce model error in each training step.
- Optimization algorithms define how this process is performed (Stochastic Gradient Descent).
- All optimization logic is encapsulated in the optimizer object.
- There are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.
- Inside the training loop, optimization happens in three steps:
  - Call *optimizer.zero_grad()* to reset the gradients of model parameters.
  - Backpropagate the prediction loss with a call to *loss.backward()*
  - *optimizer.step()* to adjust the parameters by the gradients collected in the backward pass.

Full Implementation

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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


def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

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

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


SAVE AND LOAD THE MODEL

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

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

- To load model weights, you need to create an instance of the same model first, and then load the parameters using load_state_dict() method.

```
model = models.vgg16() # we do not specify pretrained=True, i.e. do not load default weights
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

- Saving and Loading Models with Shapes

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

```
model = torch.load('model.pth')
```