tut7_NN

Oct, 2022
Hongzheng Yang

Official website: https://pytorch.org/tutorials/

[1]:
```
# pip install torch torchvision numpy matplotlib
import torch
import torchvision
from torch import nn
import torch.nn.functional as func
from torch.utils.data import DataLoader
from torchvision import datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
# Train transformation
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize((0.5), (0.5))
])
# Test transformation
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5), (0.5))
])

# Download training data from open datasets.
training_data = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
```

```
    transform=train_transform,
)

# Download test data from open datasets.
test_data = datasets.CIFAR10(
    root="data",
    train=False,
    download=True,
    transform=test_transform,
)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[3]:
```
# Training batch size
batch_size = 256

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.figure(figsize=(15,15))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Each batch tensor shape
for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)

    imshow(torchvision.utils.make_grid(X, nrow=20))

    break
```

```
Shape of X [N, C, H, W]:  torch.Size([256, 3, 32, 32])
Shape of y:  torch.Size([256]) torch.int64
```

[4]:
```python
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(3 * 32 * 32, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

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

```
model = NeuralNetwork().to(device)
# for X, y in test_dataloader:
#     output = model(X)
#     break
print(model)
```

```
Using cpu device
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3072, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

[5]:
```
# Loss function
loss_fn = nn.CrossEntropyLoss()

# SGD Optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)
```

[6]:
```
# Training function
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)

    # Turn on training mode
    model.train()
    train_loss, correct = 0, 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

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

        # record loss
        train_loss += loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
```

```
    train_loss /= len(dataloader)
    correct /= size

    print(f" Train accuracy: {(100*correct):>0.1f}%, Avg loss: {train_loss:
↪>8f}")
```

[7]:
```
# Test function
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)

    # Turn on evalution mode
    model.eval()
    test_loss, correct = 0, 0

    # Turn off gradient descent
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)

            # record loss
            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 accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f}")
```

[8]:
```
# Total training epochs
epochs = 5

for t in range(epochs):
    print('\n', "=" * 15, "Epoch", t + 1, "=" * 15)
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)

print(" Done!")
```

```
=============== Epoch 1 ===============
Train accuracy: 34.4%, Avg loss: 1.858915
Test accuracy: 41.5%, Avg loss: 1.657532

=============== Epoch 2 ===============
Train accuracy: 43.9%, Avg loss: 1.591077
Test accuracy: 45.4%, Avg loss: 1.541371
```

```
=============== Epoch 3 ===============
Train accuracy: 47.6%, Avg loss: 1.492796
Test accuracy: 48.1%, Avg loss: 1.477654

=============== Epoch 4 ===============
Train accuracy: 50.0%, Avg loss: 1.425377
Test accuracy: 49.4%, Avg loss: 1.440541

=============== Epoch 5 ===============
Train accuracy: 51.8%, Avg loss: 1.371083
Test accuracy: 50.1%, Avg loss: 1.413758
Done!
```

[9]:
```python
# Saving model weights
# torch.save(model)
torch.save(model.state_dict(), "model.pth")
print(" Saved PyTorch Model State to model.pth")
```

```
Saved PyTorch Model State to model.pth
```

[10]:
```python
# Build the network
model = NeuralNetwork()

# Load trained weights
# .pth pt pkl pth.tar
model.load_state_dict(torch.load("model.pth"))
```

[10]:
```
<All keys matched successfully>
```

[11]:
```python
# 10 Classes
classes = [
    "Airplane",
    "Automobile",
    "Bird",
    "Cat",
    "Deer",
    "Dog",
    "Frog",
    "Horse",
    "Ship",
    "Truck",
]

# Evaluation mode
model.eval()

# Get one sample
```

```
x, y = torch.tensor(test_data.data[0]).float().unsqueeze(0), test_data.
 ↪targets[0]
print(x.shape)
# x, y = test_data.data[0], test_data.targets[1]

# Turn off gradient descent
with torch.no_grad():
    pred = model(x)
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f' Predicted: "{predicted}", Actual: "{actual}"')
```

```
torch.Size([1, 32, 32, 3])
 Predicted: "Airplane", Actual: "Cat"
```