simplenet

September 8, 2024

[22]:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import time

# Define transformations for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

[23]:

```
# SimpleNet model
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.conv6 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.conv7 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(512, 1024)
        self.fc2 = nn.Linear(1024, 10)
```

```python
    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = torch.relu(self.conv3(x))
        x = self.pool(torch.relu(self.conv4(x)))
        x = torch.relu(self.conv5(x))
        x = self.pool(torch.relu(self.conv6(x)))
        x = torch.relu(self.conv7(x))

        x = x.view(x.size(0), -1)  # Flatten the feature map
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

[24]:

```python
# Initialize the model, define the loss function and the optimizer
model = SimpleNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop with progress output and model saving
def train(model, train_loader, criterion, optimizer, epochs=5,␣
 ↪save_path='simple_net.pth'):
    model.train()  # Set model to training mode
    for epoch in range(epochs):
        start_time = time.time()
        running_loss = 0.0
        for batch_idx, (images, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Clear previous gradients
            outputs = model(images)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            running_loss += loss.item()

            if batch_idx % 100 == 0:  # Print progress every 100 batches
                print(f"Epoch [{epoch+1}/{epochs}], Batch [{batch_idx}/
 ↪{len(train_loader)}], Loss: {loss.item():.4f}")

        epoch_time = time.time() - start_time
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {running_loss/
 ↪len(train_loader):.4f}, Time: {epoch_time:.2f} seconds")

    # Save the model
    torch.save(model.state_dict(), save_path)
    print(f"Model saved to {save_path}")
```

[25]:
```python
# Test loop after loading the saved model
def test(model, test_loader, save_path='simple_net.pth'):
    # Load the saved model
    model.load_state_dict(torch.load(save_path))
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')
```

[26]:
```python
# Train the model
train(model, train_loader, criterion, optimizer)
```

```
Epoch [1/5], Batch [0/938], Loss: 2.3003
Epoch [1/5], Batch [100/938], Loss: 0.6892
Epoch [1/5], Batch [200/938], Loss: 0.2214
Epoch [1/5], Batch [300/938], Loss: 0.3632
Epoch [1/5], Batch [400/938], Loss: 0.1521
Epoch [1/5], Batch [500/938], Loss: 0.0701
Epoch [1/5], Batch [600/938], Loss: 0.3734
Epoch [1/5], Batch [700/938], Loss: 0.1884
Epoch [1/5], Batch [800/938], Loss: 0.0386
Epoch [1/5], Batch [900/938], Loss: 0.0330
Epoch [1/5], Loss: 0.2977, Time: 153.24 seconds
Epoch [2/5], Batch [0/938], Loss: 0.0498
Epoch [2/5], Batch [100/938], Loss: 0.0794
Epoch [2/5], Batch [200/938], Loss: 0.0337
Epoch [2/5], Batch [300/938], Loss: 0.0334
Epoch [2/5], Batch [400/938], Loss: 0.0870
Epoch [2/5], Batch [500/938], Loss: 0.0111
Epoch [2/5], Batch [600/938], Loss: 0.1014
Epoch [2/5], Batch [700/938], Loss: 0.1090
Epoch [2/5], Batch [800/938], Loss: 0.1273
Epoch [2/5], Batch [900/938], Loss: 0.0251
Epoch [2/5], Loss: 0.0608, Time: 163.97 seconds
Epoch [3/5], Batch [0/938], Loss: 0.0324
Epoch [3/5], Batch [100/938], Loss: 0.0252
Epoch [3/5], Batch [200/938], Loss: 0.0752
Epoch [3/5], Batch [300/938], Loss: 0.0013
Epoch [3/5], Batch [400/938], Loss: 0.1943
```

```
Epoch [3/5], Batch [500/938], Loss: 0.0554
Epoch [3/5], Batch [600/938], Loss: 0.0401
Epoch [3/5], Batch [700/938], Loss: 0.0049
Epoch [3/5], Batch [800/938], Loss: 0.0668
Epoch [3/5], Batch [900/938], Loss: 0.0063
Epoch [3/5], Loss: 0.0454, Time: 169.11 seconds
Epoch [4/5], Batch [0/938], Loss: 0.0940
Epoch [4/5], Batch [100/938], Loss: 0.0004
Epoch [4/5], Batch [200/938], Loss: 0.0483
Epoch [4/5], Batch [300/938], Loss: 0.1005
Epoch [4/5], Batch [400/938], Loss: 0.0577
Epoch [4/5], Batch [500/938], Loss: 0.0027
Epoch [4/5], Batch [600/938], Loss: 0.0006
Epoch [4/5], Batch [700/938], Loss: 0.0038
Epoch [4/5], Batch [800/938], Loss: 0.0005
Epoch [4/5], Batch [900/938], Loss: 0.2587
Epoch [4/5], Loss: 0.0413, Time: 170.53 seconds
Epoch [5/5], Batch [0/938], Loss: 0.0033
Epoch [5/5], Batch [100/938], Loss: 0.0037
Epoch [5/5], Batch [200/938], Loss: 0.0414
Epoch [5/5], Batch [300/938], Loss: 0.0826
Epoch [5/5], Batch [400/938], Loss: 0.0200
Epoch [5/5], Batch [500/938], Loss: 0.0149
Epoch [5/5], Batch [600/938], Loss: 0.0091
Epoch [5/5], Batch [700/938], Loss: 0.0630
Epoch [5/5], Batch [800/938], Loss: 0.0147
Epoch [5/5], Batch [900/938], Loss: 0.1197
Epoch [5/5], Loss: 0.0348, Time: 165.35 seconds
Model saved to simple_net.pth
```

[27]:
```
# Test the model by reloading it
test(model, test_loader)
```

```
/tmp/ipykernel_2294/3840639354.py:4: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load(save_path))
```

```
Accuracy: 99.09%
```

[ ]:

```

```