Week 6 Questions (220962018)

February 20, 2025

1 Week 6 Lab Exercise

[5]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms , models
import os
```

1.1 Lab Exercise

[6]:
```
# Make sure the ModelFiles directory exists
if not os.path.exists('./ModelFiles'):
    os.makedirs('./ModelFiles')


# Define the CNN model
class CNNClassifier(nn.Module):
    def __init__(self):
        super(CNNClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)  # 10 classes for MNIST

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 64 * 7 * 7)  # Flatten the tensor
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# Load MNIST dataset
```

```python
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,))])
trainset = datasets.MNIST(root='./data', train=True, download=True,␣
↪transform=transform)
testset = datasets.MNIST(root='./data', train=False, download=True,␣
↪transform=transform)

trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
testloader = DataLoader(testset, batch_size=64, shuffle=False)

# Initialize the model
model = CNNClassifier()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f"Epoch {epoch + 1}, Loss: {running_loss / len(trainloader):.4f},␣
↪Accuracy: {100 * correct / total:.2f}%")

# Save the trained model
torch.save(model.state_dict(), "./ModelFiles/model.pt")
print("Model saved to './ModelFiles/model.pt'")
```

```
Epoch 1, Loss: 0.1632, Accuracy: 95.01%
Epoch 2, Loss: 0.0475, Accuracy: 98.56%
Epoch 3, Loss: 0.0328, Accuracy: 98.96%
Epoch 4, Loss: 0.0255, Accuracy: 99.20%
Epoch 5, Loss: 0.0178, Accuracy: 99.44%
Model saved to './ModelFiles/model.pt'
```

[7]:
```python
# Define the CNN model (same as MNIST_CNN.py)
class CNNClassifier(nn.Module):
    def __init__(self):
        super(CNNClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)  # 10 classes for FashionMNIST

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 64 * 7 * 7)  # Flatten the tensor
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Load FashionMNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
 ↪5,), (0.5,))])
fashion_mnist_testset = datasets.FashionMNIST(root='./data', train=False,␣
 ↪download=True, transform=transform)
fashion_mnist_trainset = datasets.FashionMNIST(root='./data', train=True,␣
 ↪download=True, transform=transform)

test_loader = DataLoader(fashion_mnist_testset, batch_size=64, shuffle=False)

# Load the pre-trained model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNClassifier()

# Load the pre-trained weights (MNIST model)
model.load_state_dict(torch.load("./ModelFiles/model.pt"))

# Move the model to device
model.to(device)

# Set the model to evaluation mode
```

```
model.eval()

# Evaluate the model on FashionMNIST
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Accuracy on FashionMNIST test data using pre-trained MNIST model:␣
 ↪{accuracy:.2f}%")
```

```
Accuracy on FashionMNIST test data using pre-trained MNIST model: 7.54%
```

[8]:
```
# Define image transformations for training and validation
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),  # Ensure images are resized to 224x224␣
 ↪(fixed size)
    transforms.RandomHorizontalFlip(),  # Data augmentation (flipping images)
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])␣
 ↪ # Normalization for AlexNet
])

validation_transforms = transforms.Compose([
    transforms.Resize((224, 224)),  # Ensure images are resized to 224x224␣
 ↪(fixed size)
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])␣
 ↪ # Normalization for AlexNet
])

# Load the datasets
train_dir = './cats_and_dogs_filtered/train'
val_dir = './cats_and_dogs_filtered/validation'

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
val_data = datasets.ImageFolder(val_dir, transform=validation_transforms)

# Load data into DataLoader
batch_size = 32
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

```python
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)

# Load the pre-trained AlexNet model
alexnet = models.alexnet(weights="IMAGENET1K_V1")  # Using the pretrained weights

# Freeze all layers so no weights are updated except for the final classifier
for param in alexnet.parameters():
    param.requires_grad = False

# Modify the final layer (classifier) to have two outputs (for cats and dogs)
alexnet.classifier[6] = nn.Linear(in_features=4096, out_features=2)

# Move model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
alexnet.to(device)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(alexnet.classifier[6].parameters(), lr=0.001)  # Only train the last layer

# Train the model
num_epochs = 5
for epoch in range(num_epochs):
    alexnet.train()  # Set the model to training mode
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = alexnet(inputs)
        loss = criterion(outputs, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Calculate accuracy
```

```
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct / total
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Accuracy:␣
↪{epoch_accuracy:.2f}%")

# Evaluate the model
alexnet.eval()  # Set the model to evaluation mode
correct = 0
total = 0

with torch.no_grad():
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        outputs = alexnet(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

validation_accuracy = 100 * correct / total
print(f"Validation Accuracy: {validation_accuracy:.2f}%")

# Save the model
torch.save(alexnet.state_dict(), 'alexnet_no_finetuning_cats_and_dogs.pth')
```

```
Epoch 1/5, Loss: 0.2125, Accuracy: 90.65%
Epoch 2/5, Loss: 0.0973, Accuracy: 96.45%
Epoch 3/5, Loss: 0.0801, Accuracy: 96.55%
Epoch 4/5, Loss: 0.0694, Accuracy: 97.35%
Epoch 5/5, Loss: 0.0498, Accuracy: 98.30%
Validation Accuracy: 94.90%
```

[11]:

```
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define the CNN model
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)  # Padding to␣
↪maintain spatial dimensions
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # Padding to␣
↪maintain spatial dimensions
```

```python
        self.fc1 = nn.Linear(self._get_conv_output_size(), 128)  # Dynamically␣
↪calculate the input size for fc1
        self.fc2 = nn.Linear(128, 10)

    def _get_conv_output_size(self):
        # Simulate a single forward pass to determine the output size of conv␣
↪layers
        x = torch.zeros(1, 1, 28, 28)  # Size of input image (28x28x1 for MNIST)
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)  # Pooling after conv1
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)  # Pooling after conv2
        return x.numel()  # Get the number of elements in the output tensor␣
↪after conv layers

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)  # Flatten the output
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# Define transformations and load data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,))])
train_data = datasets.MNIST(root='./data', train=True, download=True,␣
↪transform=transform)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

# Create model and optimizer
model = CNNModel().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()


# Function to load checkpoint
def load_checkpoint(filename="./checkpoints/checkpoint.pt"):
    checkpoint = torch.load(filename)
    model.load_state_dict(checkpoint["model_state"])
    optimizer.load_state_dict(checkpoint["optimizer_state"])
    epoch = checkpoint["last_epoch"]
    loss = checkpoint["last_loss"]
    print(f"Checkpoint loaded from epoch {epoch}, loss: {loss:.4f}")
```

```python
    return epoch, loss


# Function to save checkpoint
def save_checkpoint(epoch, model, optimizer, loss, filename="./checkpoints/
 ↪checkpoint.pt"):
    if not os.path.exists('./checkpoints'):
        os.makedirs('./checkpoints')

    checkpoint = {
        "last_epoch": epoch,
        "model_state": model.state_dict(),
        "optimizer_state": optimizer.state_dict(),
        "last_loss": loss
    }
    torch.save(checkpoint, filename)
    print(f"Checkpoint saved at epoch {epoch}")


# Load checkpoint and resume training
checkpoint_file = './checkpoints/checkpoint.pt'
start_epoch = 0
if os.path.exists(checkpoint_file):
    start_epoch, last_loss = load_checkpoint(checkpoint_file)
else:
    print("No checkpoint found, starting from scratch.")

# Resume training from the checkpoint
num_epochs = 3
for epoch in range(start_epoch, num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass
        loss.backward()
        optimizer.step()
```

```
        running_loss += loss.item()

        # Calculate accuracy
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    avg_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct / total
    print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_loss:.4f}, Accuracy:␣
↪{epoch_accuracy:.2f}%")

    # Optionally save the checkpoint again after every epoch or based on␣
↪conditions
    save_checkpoint(epoch, model, optimizer, avg_loss)

print("Resumed training finished.")
```

```
Checkpoint loaded from epoch 1, loss: 0.0451
Epoch [2/3], Loss: 0.0324, Accuracy: 98.98%
Checkpoint saved at epoch 1
Epoch [3/3], Loss: 0.0226, Accuracy: 99.31%
Checkpoint saved at epoch 2
Resumed training finished.
```