hw3pdl

April 14, 2025

[34]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import time
from torch.utils.data import DataLoader
```

- In this first cell, I imported all the necessary libraries for training and evaluating my CNN. This includes PyTorch for model building and torchvision for data handling.

[35]:
```
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),])

# Training and Testing here
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,␣
 ↪transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_test)

trainload = DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)
testload = DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)

classes = trainset.classes
```

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

- For this cell, I fetched CIFAR10 dataset and divided into training and testing sets. Also, I defined batch size of 128 here and did normalization which helps the model train faster and more stably.

Here I did some changes in the baseline model and updated for all the type which is listed below:

- conv1: 32 filters
- conv2: 64 filters
- fc1: 256 fully connected neurons
- fc2: 128 fully connected neurons

[36]:
```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(64 * 5 * 5, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- In this cell, I defined my updated baseline LeNet-5 model. I increased the number of filters and used max pooling instead of average pooling. This setup helps extract stronger and more varied features from the input images.

[37]:
```
class LeNet5Dropout(nn.Module):
    def __init__(self):
        super(LeNet5Dropout, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(64 * 5 * 5, 256)
        self.dropout = nn.Dropout(0.3)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
```

```
        x = self.fc3(x)
        return x
```

- This cell defines my LeNet-5 model with a dropout layer. I applied dropout after the first fully connected layer to help prevent overfitting during training. My Dropout rate is 0.3 here.

[38]:
```
class LeNet5BatchNorm(nn.Module):
    def __init__(self):
        super(LeNet5BatchNorm, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.bn2 = nn.BatchNorm2d(64)
        self.fc1 = nn.Linear(64 * 5 * 5, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = x.view(-1, 64 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- Here, I created a version of LeNet-5 that uses batch normalization after each convolutional layer. This improves training stability and speeds up convergence.

[39]:
```
def train_eval(model, epochs=10, lr=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    crt = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    start_time = time.time()

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in trainload:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = crt(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
            running_loss += loss.item()

        print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainload):.4f}")

    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testload:
            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()

    end_time = time.time()
    print(f"Test accuracy: {100 * correct / total:.2f}%")
    print(f"Training time: {end_time - start_time:.2f} seconds")
```

- This cell contains the training and evaluation function. It handles model training over multiple epochs, computes loss, and reports final accuracy and training time.

[40]:
```
print("Baseline LeNet-5")
baseline = LeNet5()
train_eval(baseline)

print("\nLeNet-5 with Dropout")
dropout = LeNet5Dropout()
train_eval(dropout)

print("\nLeNet-5 with BatchNorm")
batchnorm = LeNet5BatchNorm()
train_eval(batchnorm)
```

```
Baseline LeNet-5
Epoch 1, Loss: 1.5079
Epoch 2, Loss: 1.1066
Epoch 3, Loss: 0.9394
Epoch 4, Loss: 0.8157
Epoch 5, Loss: 0.7130
Epoch 6, Loss: 0.6284
Epoch 7, Loss: 0.5473
Epoch 8, Loss: 0.4771
Epoch 9, Loss: 0.4112
Epoch 10, Loss: 0.3387
Test accuracy: 72.02%
Training time: 73.85 seconds
```

```
LeNet-5 with Dropout
Epoch 1, Loss: 1.5789
Epoch 2, Loss: 1.2087
Epoch 3, Loss: 1.0334
Epoch 4, Loss: 0.9246
Epoch 5, Loss: 0.8333
Epoch 6, Loss: 0.7607
Epoch 7, Loss: 0.6934
Epoch 8, Loss: 0.6431
Epoch 9, Loss: 0.5946
Epoch 10, Loss: 0.5447
Test accuracy: 72.38%
Training time: 73.97 seconds

LeNet-5 with BatchNorm
Epoch 1, Loss: 1.3263
Epoch 2, Loss: 0.9804
Epoch 3, Loss: 0.8290
Epoch 4, Loss: 0.7209
Epoch 5, Loss: 0.6446
Epoch 6, Loss: 0.5696
Epoch 7, Loss: 0.5092
Epoch 8, Loss: 0.4532
Epoch 9, Loss: 0.3943
Epoch 10, Loss: 0.3533
Test accuracy: 74.57%
Training time: 73.26 seconds
```

- In the last cell I called all LeNet-5 model's versions: Baseline, With Dropout, and with Batch normalization. This helps me compare their effectiveness against the baseline.

1 Final result

- After running each model for 10 epochs, it is stated that batch normalization model has the highest testing accuracy among the all then comes with Dropput and Baseline model.

[ ]: