homework2

February 18, 2025

[4]:

```python
## AlexNet -> CIFAR-10



import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import numpy as np

#NUM_EPOCHS
num_epochs = 20

#  AlexNet model adapted for CIFAR-10
class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            # Conv1: input [3, 32, 32] -> output [64, 32, 32]
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # output: [64, 16, 16]

            # Conv2: output [192, 16, 16]
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # output: [192, 8, 8]

            # Conv3: output [384, 8, 8]
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            # Conv4: output [256, 8, 8]
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
```

```
            nn.ReLU(inplace=True),

            # Conv5: output [256, 8, 8] then pool to [256, 4, 4]
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Classifier: Adjusted for a feature map size of 4x4 after pooling.
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 4 * 4, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
])

# CIFAR-10 Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
 ↪download=True, transform=transform_test)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AlexNet(num_classes=10).to(device)

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

# Lists to store metrics
train_losses = []
train_accs = []
test_losses = []
test_accs = []

# Training loop
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    # Training phase
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], Loss: {running_loss/100:.4f}')
            running_loss = 0.0

    # Calculate epoch training metrics
    epoch_train_loss = running_loss / len(trainloader)
```

```
    epoch_train_acc = 100 * correct / total
    train_losses.append(epoch_train_loss)
    train_accs.append(epoch_train_acc)

    # Validation phase
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Calculate epoch validation metrics
    epoch_test_loss = val_loss / len(testloader)
    epoch_test_acc = 100 * correct / total
    test_losses.append(epoch_test_loss)
    test_accs.append(epoch_test_acc)

    print(f'Epoch [{epoch+1}/{num_epochs}]:')
    print(f'Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.
↪2f}%')
    print(f'Test Loss: {epoch_test_loss:.4f}, Test Acc: {epoch_test_acc:.2f}%')

print('Finished Training')

# Count and print total parameters
total_params = sum(p.numel() for p in model.parameters())
print(f'\nTotal number of parameters: {total_params:,}')

# Count trainable parameters
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Trainable parameters: {trainable_params:,}')


# Plot the curves
plt.figure(figsize=(15, 5))

# Loss curves
plt.subplot(1, 2, 1)
```

```python
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()

# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()

# Compute and plot confusion matrix
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(10, 10))
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()
```

```python
# Calculate precision, recall, f1-score for each class
from sklearn.metrics import precision_recall_fscore_support, accuracy_score

# Calculate metrics
precision, recall, f1, _ = precision_recall_fscore_support(all_labels, all_preds, average=None)
accuracy = accuracy_score(all_labels, all_preds)

# Print overall accuracy
print(f"\nOverall Accuracy: {accuracy*100:.2f}%\n")

# Print per-class metrics
print("Per-class metrics:")
print("Class\t\tPrecision\tRecall\t\tF1-Score")
print("-" * 60)
for i in range(len(classes)):
    print(f"{classes[i]:<12}\t{precision[i]*100:>8.2f}%\t{recall[i]*100:>8.2f}%\t{f1[i]*100:>8.2f}%")

# Print macro-averaged metrics
macro_precision = precision.mean()
macro_recall = recall.mean()
macro_f1 = f1.mean()

print("\nMacro-averaged metrics:")
print(f"Precision: {macro_precision*100:.2f}%")
print(f"Recall: {macro_recall*100:.2f}%")
print(f"F1-Score: {macro_f1*100:.2f}%")
```

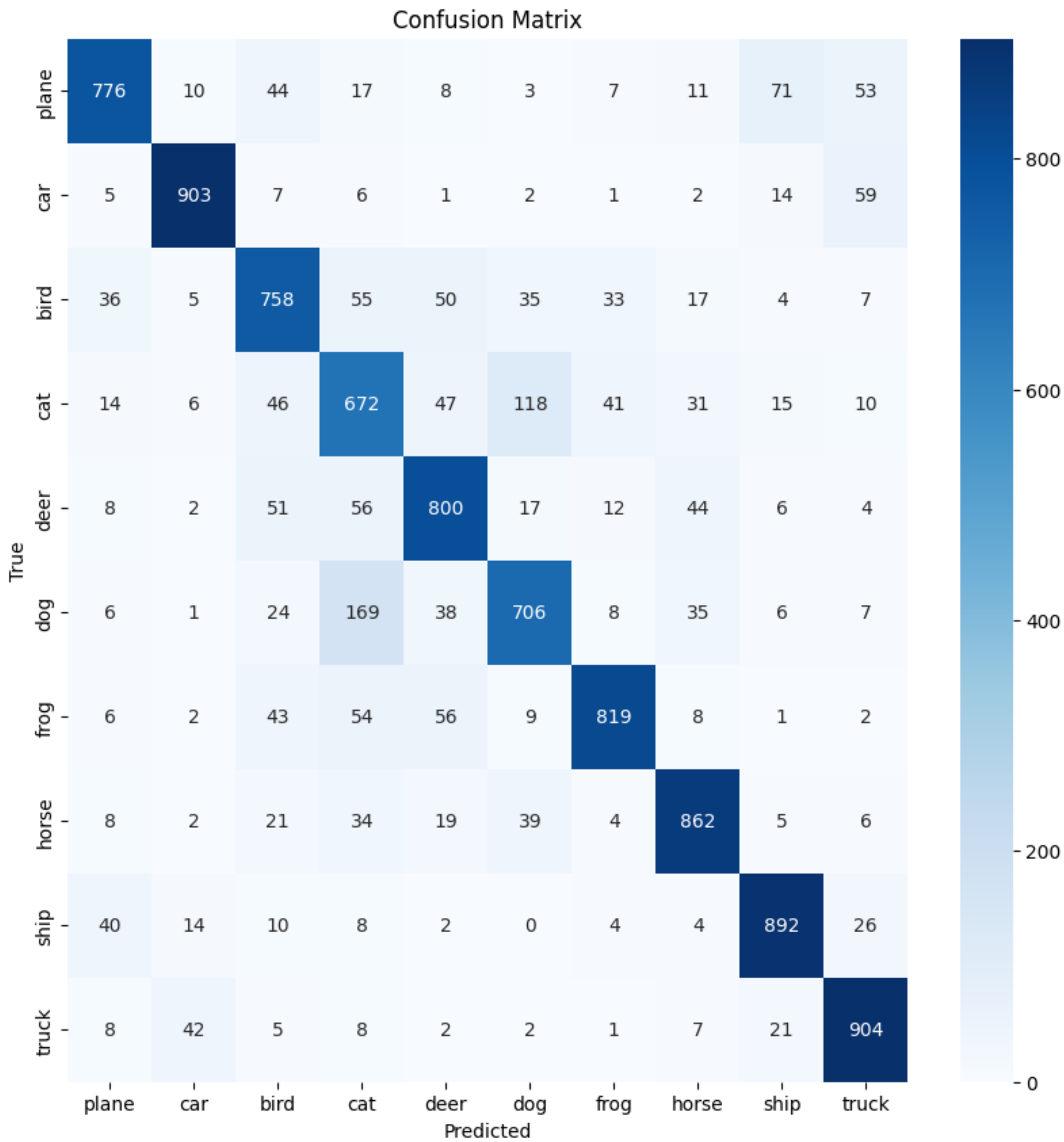
```
Epoch [1/20], Step [100/391], Loss: 2.0382
Epoch [1/20], Step [200/391], Loss: 1.7828
Epoch [1/20], Step [300/391], Loss: 1.6285
Epoch [1/20]:
Train Loss: 0.3568, Train Acc: 33.38%
Test Loss: 1.4139, Test Acc: 48.30%
Epoch [2/20], Step [100/391], Loss: 1.4378
Epoch [2/20], Step [200/391], Loss: 1.3744
Epoch [2/20], Step [300/391], Loss: 1.3306
Epoch [2/20]:
Train Loss: 0.3021, Train Acc: 50.04%
Test Loss: 1.2504, Test Acc: 54.59%
Epoch [3/20], Step [100/391], Loss: 1.2388
Epoch [3/20], Step [200/391], Loss: 1.2057
Epoch [3/20], Step [300/391], Loss: 1.1444
Epoch [3/20]:
Train Loss: 0.2580, Train Acc: 58.02%
Test Loss: 1.0903, Test Acc: 60.54%
```

```
Epoch [4/20], Step [100/391], Loss: 1.0663
Epoch [4/20], Step [200/391], Loss: 1.0528
Epoch [4/20], Step [300/391], Loss: 1.0214
Epoch [4/20]:
Train Loss: 0.2356, Train Acc: 63.26%
Test Loss: 0.9563, Test Acc: 66.35%
Epoch [5/20], Step [100/391], Loss: 0.9517
Epoch [5/20], Step [200/391], Loss: 0.9553
Epoch [5/20], Step [300/391], Loss: 0.9548
Epoch [5/20]:
Train Loss: 0.2157, Train Acc: 66.93%
Test Loss: 0.8422, Test Acc: 70.51%
Epoch [6/20], Step [100/391], Loss: 0.8897
Epoch [6/20], Step [200/391], Loss: 0.8886
Epoch [6/20], Step [300/391], Loss: 0.8805
Epoch [6/20]:
Train Loss: 0.2043, Train Acc: 69.36%
Test Loss: 0.8423, Test Acc: 70.52%
Epoch [7/20], Step [100/391], Loss: 0.8480
Epoch [7/20], Step [200/391], Loss: 0.8317
Epoch [7/20], Step [300/391], Loss: 0.8230
Epoch [7/20]:
Train Loss: 0.1942, Train Acc: 71.16%
Test Loss: 0.8134, Test Acc: 71.71%
Epoch [8/20], Step [100/391], Loss: 0.7967
Epoch [8/20], Step [200/391], Loss: 0.7870
Epoch [8/20], Step [300/391], Loss: 0.7681
Epoch [8/20]:
Train Loss: 0.1851, Train Acc: 73.11%
Test Loss: 0.7466, Test Acc: 74.62%
Epoch [9/20], Step [100/391], Loss: 0.7534
Epoch [9/20], Step [200/391], Loss: 0.7505
Epoch [9/20], Step [300/391], Loss: 0.7576
Epoch [9/20]:
Train Loss: 0.1755, Train Acc: 74.11%
Test Loss: 0.6959, Test Acc: 76.41%
Epoch [10/20], Step [100/391], Loss: 0.7171
Epoch [10/20], Step [200/391], Loss: 0.7216
Epoch [10/20], Step [300/391], Loss: 0.7242
Epoch [10/20]:
Train Loss: 0.1725, Train Acc: 75.25%
Test Loss: 0.7191, Test Acc: 75.62%
Epoch [11/20], Step [100/391], Loss: 0.6978
Epoch [11/20], Step [200/391], Loss: 0.7018
Epoch [11/20], Step [300/391], Loss: 0.6982
Epoch [11/20]:
Train Loss: 0.1604, Train Acc: 76.35%
Test Loss: 0.6960, Test Acc: 76.15%
```

```
Epoch [12/20], Step [100/391], Loss: 0.6675
Epoch [12/20], Step [200/391], Loss: 0.6658
Epoch [12/20], Step [300/391], Loss: 0.6781
Epoch [12/20]:
Train Loss: 0.1581, Train Acc: 77.11%
Test Loss: 0.6943, Test Acc: 76.15%
Epoch [13/20], Step [100/391], Loss: 0.6526
Epoch [13/20], Step [200/391], Loss: 0.6572
Epoch [13/20], Step [300/391], Loss: 0.6717
Epoch [13/20]:
Train Loss: 0.1557, Train Acc: 77.43%
Test Loss: 0.6363, Test Acc: 78.59%
Epoch [14/20], Step [100/391], Loss: 0.6329
Epoch [14/20], Step [200/391], Loss: 0.6285
Epoch [14/20], Step [300/391], Loss: 0.6486
Epoch [14/20]:
Train Loss: 0.1559, Train Acc: 78.16%
Test Loss: 0.6177, Test Acc: 78.86%
Epoch [15/20], Step [100/391], Loss: 0.6116
Epoch [15/20], Step [200/391], Loss: 0.6161
Epoch [15/20], Step [300/391], Loss: 0.6205
Epoch [15/20]:
Train Loss: 0.1435, Train Acc: 78.99%
Test Loss: 0.6189, Test Acc: 79.29%
Epoch [16/20], Step [100/391], Loss: 0.6023
Epoch [16/20], Step [200/391], Loss: 0.6187
Epoch [16/20], Step [300/391], Loss: 0.6028
Epoch [16/20]:
Train Loss: 0.1394, Train Acc: 79.35%
Test Loss: 0.6034, Test Acc: 79.40%
Epoch [17/20], Step [100/391], Loss: 0.5899
Epoch [17/20], Step [200/391], Loss: 0.5943
Epoch [17/20], Step [300/391], Loss: 0.6007
Epoch [17/20]:
Train Loss: 0.1411, Train Acc: 79.67%
Test Loss: 0.6417, Test Acc: 78.27%
Epoch [18/20], Step [100/391], Loss: 0.5733
Epoch [18/20], Step [200/391], Loss: 0.5586
Epoch [18/20], Step [300/391], Loss: 0.5919
Epoch [18/20]:
Train Loss: 0.1408, Train Acc: 80.19%
Test Loss: 0.6284, Test Acc: 78.90%
Epoch [19/20], Step [100/391], Loss: 0.5645
Epoch [19/20], Step [200/391], Loss: 0.5689
Epoch [19/20], Step [300/391], Loss: 0.5870
Epoch [19/20]:
Train Loss: 0.1356, Train Acc: 80.61%
Test Loss: 0.5892, Test Acc: 80.16%
```

```
Epoch [20/20], Step [100/391], Loss: 0.5393
Epoch [20/20], Step [200/391], Loss: 0.5727
Epoch [20/20], Step [300/391], Loss: 0.5372
Epoch [20/20]:
Train Loss: 0.1321, Train Acc: 81.09%
Test Loss: 0.5591, Test Acc: 80.92%
Finished Training

Total number of parameters: 35,855,178
Trainable parameters: 35,855,178
```



[1]:
```
## AlexNet -> CIFAR-100

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
```

```
import seaborn as sns
import numpy as np

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# AlexNet- CIFAR-100
class AlexNet(nn.Module):
    def __init__(self, num_classes=100):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            # Conv1: input [3, 32, 32] -> output [64, 32, 32]
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # output: [64, 16, 16]

            # Conv2: output [192, 16, 16]
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.BatchNorm2d(192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # output: [192, 8, 8]

            # Conv3: output [384, 8, 8]
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(inplace=True),

            # Conv4: output [256, 8, 8]
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),

            # Conv5: output [256, 8, 8] then pool to [256, 4, 4]
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Classifier: Adjusted for a feature map size of 4x4 after pooling.
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(256 * 4 * 4, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
```

```python
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

# CIFAR-100 Dataset
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR100(root='./data', train=False,
 ↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,
 ↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AlexNet(num_classes=100).to(device)

# Training loop
## NUM EPOCHS
num_epochs = 50

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,
 ↪weight_decay=5e-4)

# Learning rate scheduler
```

```
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
 ↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
 ↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
```

```
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')

print(f'Best Test Accuracy: {best_acc:.2f}%')

# Compute confusion matrix
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()
```

```python
# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()

# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

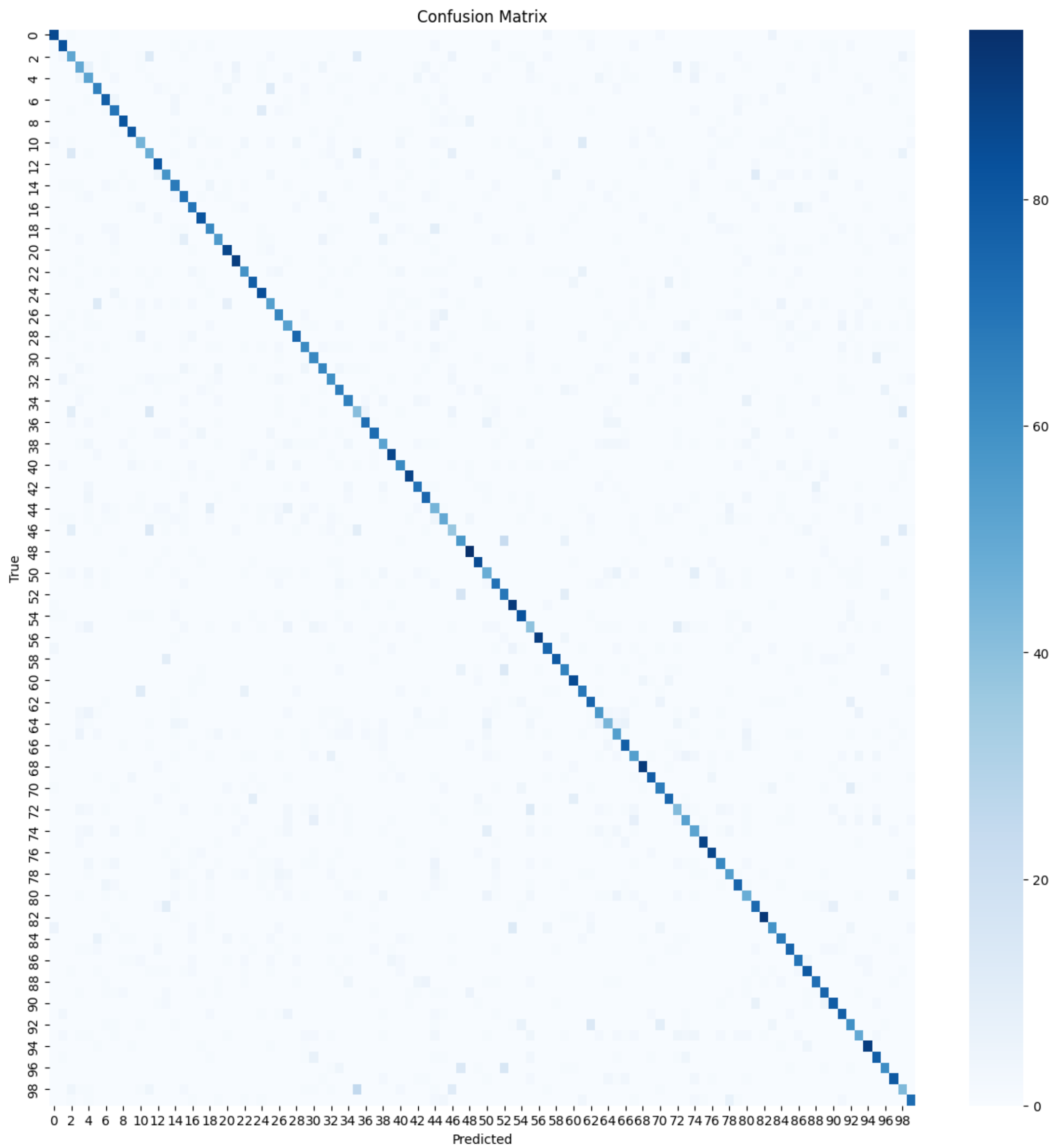
```
Epoch [1/50], Step [100/391], Loss: 4.3987, Acc: 3.55%
Epoch [1/50], Step [200/391], Loss: 3.9365, Acc: 6.06%
Epoch [1/50], Step [300/391], Loss: 3.6848, Acc: 8.01%
Epoch [1/50] Test Accuracy: 17.03%
Epoch [2/50], Step [100/391], Loss: 3.3500, Acc: 18.34%
Epoch [2/50], Step [200/391], Loss: 3.2556, Acc: 18.97%
Epoch [2/50], Step [300/391], Loss: 3.0884, Acc: 19.90%
Epoch [2/50] Test Accuracy: 28.93%
Epoch [3/50], Step [100/391], Loss: 2.8651, Acc: 26.86%
Epoch [3/50], Step [200/391], Loss: 2.8491, Acc: 27.10%
Epoch [3/50], Step [300/391], Loss: 2.7444, Acc: 27.79%
Epoch [3/50] Test Accuracy: 30.94%
Epoch [4/50], Step [100/391], Loss: 2.5746, Acc: 33.27%
Epoch [4/50], Step [200/391], Loss: 2.5471, Acc: 33.38%
Epoch [4/50], Step [300/391], Loss: 2.4992, Acc: 33.71%
Epoch [4/50] Test Accuracy: 34.41%
Epoch [5/50], Step [100/391], Loss: 2.3609, Acc: 36.74%
Epoch [5/50], Step [200/391], Loss: 2.3617, Acc: 36.88%
Epoch [5/50], Step [300/391], Loss: 2.3202, Acc: 37.31%
Epoch [5/50] Test Accuracy: 42.25%
Epoch [6/50], Step [100/391], Loss: 2.2020, Acc: 40.56%
Epoch [6/50], Step [200/391], Loss: 2.1597, Acc: 40.97%
Epoch [6/50], Step [300/391], Loss: 2.1880, Acc: 40.94%
Epoch [6/50] Test Accuracy: 44.82%
```

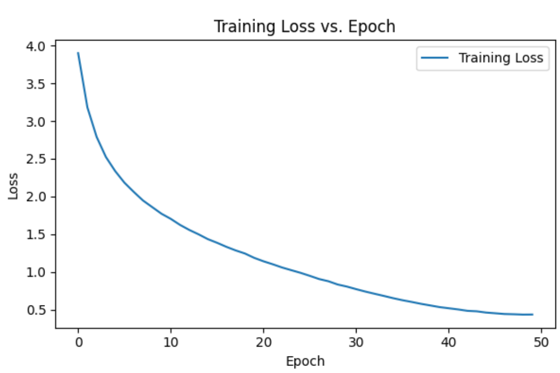
```
Epoch [7/50], Step [100/391], Loss: 2.0735, Acc: 43.39%
Epoch [7/50], Step [200/391], Loss: 2.0743, Acc: 43.48%
Epoch [7/50], Step [300/391], Loss: 2.0427, Acc: 43.66%
Epoch [7/50] Test Accuracy: 47.07%
Epoch [8/50], Step [100/391], Loss: 1.9812, Acc: 45.57%
Epoch [8/50], Step [200/391], Loss: 1.9329, Acc: 46.18%
Epoch [8/50], Step [300/391], Loss: 1.9291, Acc: 46.36%
Epoch [8/50] Test Accuracy: 44.18%
Epoch [9/50], Step [100/391], Loss: 1.8696, Acc: 48.70%
Epoch [9/50], Step [200/391], Loss: 1.8719, Acc: 48.70%
Epoch [9/50], Step [300/391], Loss: 1.8481, Acc: 48.69%
Epoch [9/50] Test Accuracy: 49.72%
Epoch [10/50], Step [100/391], Loss: 1.7607, Acc: 50.91%
Epoch [10/50], Step [200/391], Loss: 1.7647, Acc: 50.77%
Epoch [10/50], Step [300/391], Loss: 1.7802, Acc: 50.85%
Epoch [10/50] Test Accuracy: 50.58%
Epoch [11/50], Step [100/391], Loss: 1.7070, Acc: 51.59%
Epoch [11/50], Step [200/391], Loss: 1.6909, Acc: 52.01%
Epoch [11/50], Step [300/391], Loss: 1.6963, Acc: 52.20%
Epoch [11/50] Test Accuracy: 51.88%
Epoch [12/50], Step [100/391], Loss: 1.6002, Acc: 55.29%
Epoch [12/50], Step [200/391], Loss: 1.6311, Acc: 54.40%
Epoch [12/50], Step [300/391], Loss: 1.6391, Acc: 54.18%
Epoch [12/50] Test Accuracy: 53.92%
Epoch [13/50], Step [100/391], Loss: 1.5481, Acc: 55.44%
Epoch [13/50], Step [200/391], Loss: 1.5353, Acc: 55.77%
Epoch [13/50], Step [300/391], Loss: 1.5565, Acc: 55.65%
Epoch [13/50] Test Accuracy: 54.98%
Epoch [14/50], Step [100/391], Loss: 1.4545, Acc: 57.92%
Epoch [14/50], Step [200/391], Loss: 1.5090, Acc: 57.34%
Epoch [14/50], Step [300/391], Loss: 1.5061, Acc: 57.25%
Epoch [14/50] Test Accuracy: 56.42%
Epoch [15/50], Step [100/391], Loss: 1.4153, Acc: 59.14%
Epoch [15/50], Step [200/391], Loss: 1.4281, Acc: 58.98%
Epoch [15/50], Step [300/391], Loss: 1.4237, Acc: 58.97%
Epoch [15/50] Test Accuracy: 54.86%
Epoch [16/50], Step [100/391], Loss: 1.3751, Acc: 60.39%
Epoch [16/50], Step [200/391], Loss: 1.4012, Acc: 59.80%
Epoch [16/50], Step [300/391], Loss: 1.3659, Acc: 59.98%
Epoch [16/50] Test Accuracy: 57.21%
Epoch [17/50], Step [100/391], Loss: 1.2971, Acc: 62.05%
Epoch [17/50], Step [200/391], Loss: 1.3434, Acc: 61.58%
Epoch [17/50], Step [300/391], Loss: 1.3396, Acc: 61.42%
Epoch [17/50] Test Accuracy: 57.19%
Epoch [18/50], Step [100/391], Loss: 1.2652, Acc: 62.62%
Epoch [18/50], Step [200/391], Loss: 1.3002, Acc: 62.25%
Epoch [18/50], Step [300/391], Loss: 1.2702, Acc: 62.49%
Epoch [18/50] Test Accuracy: 59.15%
```

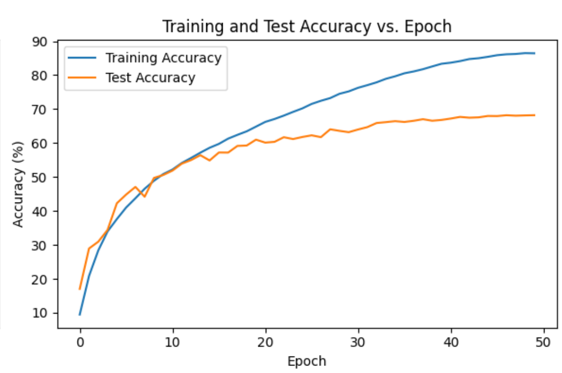
```
Epoch [19/50], Step [100/391], Loss: 1.2280, Acc: 63.28%
Epoch [19/50], Step [200/391], Loss: 1.2310, Acc: 63.65%
Epoch [19/50], Step [300/391], Loss: 1.2600, Acc: 63.40%
Epoch [19/50] Test Accuracy: 59.26%
Epoch [20/50], Step [100/391], Loss: 1.1539, Acc: 65.71%
Epoch [20/50], Step [200/391], Loss: 1.2005, Acc: 65.01%
Epoch [20/50], Step [300/391], Loss: 1.1888, Acc: 64.85%
Epoch [20/50] Test Accuracy: 60.99%
Epoch [21/50], Step [100/391], Loss: 1.1335, Acc: 66.27%
Epoch [21/50], Step [200/391], Loss: 1.1271, Acc: 66.57%
Epoch [21/50], Step [300/391], Loss: 1.1475, Acc: 66.36%
Epoch [21/50] Test Accuracy: 60.12%
Epoch [22/50], Step [100/391], Loss: 1.0717, Acc: 67.88%
Epoch [22/50], Step [200/391], Loss: 1.1049, Acc: 67.36%
Epoch [22/50], Step [300/391], Loss: 1.1032, Acc: 67.28%
Epoch [22/50] Test Accuracy: 60.35%
Epoch [23/50], Step [100/391], Loss: 1.0426, Acc: 68.11%
Epoch [23/50], Step [200/391], Loss: 1.0533, Acc: 68.25%
Epoch [23/50], Step [300/391], Loss: 1.0573, Acc: 68.24%
Epoch [23/50] Test Accuracy: 61.71%
Epoch [24/50], Step [100/391], Loss: 0.9991, Acc: 69.88%
Epoch [24/50], Step [200/391], Loss: 1.0102, Acc: 69.63%
Epoch [24/50], Step [300/391], Loss: 1.0356, Acc: 69.29%
Epoch [24/50] Test Accuracy: 61.17%
Epoch [25/50], Step [100/391], Loss: 0.9706, Acc: 70.47%
Epoch [25/50], Step [200/391], Loss: 0.9841, Acc: 70.29%
Epoch [25/50], Step [300/391], Loss: 0.9842, Acc: 70.42%
Epoch [25/50] Test Accuracy: 61.78%
Epoch [26/50], Step [100/391], Loss: 0.9342, Acc: 71.81%
Epoch [26/50], Step [200/391], Loss: 0.9563, Acc: 71.57%
Epoch [26/50], Step [300/391], Loss: 0.9453, Acc: 71.62%
Epoch [26/50] Test Accuracy: 62.27%
Epoch [27/50], Step [100/391], Loss: 0.8741, Acc: 73.02%
Epoch [27/50], Step [200/391], Loss: 0.8968, Acc: 72.84%
Epoch [27/50], Step [300/391], Loss: 0.9130, Acc: 72.62%
Epoch [27/50] Test Accuracy: 61.73%
Epoch [28/50], Step [100/391], Loss: 0.8707, Acc: 73.49%
Epoch [28/50], Step [200/391], Loss: 0.8632, Acc: 73.62%
Epoch [28/50], Step [300/391], Loss: 0.8705, Acc: 73.49%
Epoch [28/50] Test Accuracy: 64.04%
Epoch [29/50], Step [100/391], Loss: 0.8171, Acc: 75.14%
Epoch [29/50], Step [200/391], Loss: 0.8304, Acc: 74.78%
Epoch [29/50], Step [300/391], Loss: 0.8307, Acc: 74.64%
Epoch [29/50] Test Accuracy: 63.58%
Epoch [30/50], Step [100/391], Loss: 0.7724, Acc: 76.12%
Epoch [30/50], Step [200/391], Loss: 0.7977, Acc: 75.68%
Epoch [30/50], Step [300/391], Loss: 0.8086, Acc: 75.50%
Epoch [30/50] Test Accuracy: 63.19%
```

```
Epoch [31/50], Step [100/391], Loss: 0.7497, Acc: 77.18%
Epoch [31/50], Step [200/391], Loss: 0.7731, Acc: 76.68%
Epoch [31/50], Step [300/391], Loss: 0.7682, Acc: 76.54%
Epoch [31/50] Test Accuracy: 63.98%
Epoch [32/50], Step [100/391], Loss: 0.7076, Acc: 77.73%
Epoch [32/50], Step [200/391], Loss: 0.7343, Acc: 77.27%
Epoch [32/50], Step [300/391], Loss: 0.7485, Acc: 77.25%
Epoch [32/50] Test Accuracy: 64.68%
Epoch [33/50], Step [100/391], Loss: 0.6956, Acc: 78.27%
Epoch [33/50], Step [200/391], Loss: 0.6989, Acc: 78.24%
Epoch [33/50], Step [300/391], Loss: 0.7320, Acc: 77.82%
Epoch [33/50] Test Accuracy: 65.89%
Epoch [34/50], Step [100/391], Loss: 0.6656, Acc: 79.61%
Epoch [34/50], Step [200/391], Loss: 0.6860, Acc: 79.07%
Epoch [34/50], Step [300/391], Loss: 0.6842, Acc: 78.93%
Epoch [34/50] Test Accuracy: 66.15%
Epoch [35/50], Step [100/391], Loss: 0.6307, Acc: 80.41%
Epoch [35/50], Step [200/391], Loss: 0.6565, Acc: 79.90%
Epoch [35/50], Step [300/391], Loss: 0.6520, Acc: 79.77%
Epoch [35/50] Test Accuracy: 66.44%
Epoch [36/50], Step [100/391], Loss: 0.6190, Acc: 80.83%
Epoch [36/50], Step [200/391], Loss: 0.6147, Acc: 80.75%
Epoch [36/50], Step [300/391], Loss: 0.6274, Acc: 80.71%
Epoch [36/50] Test Accuracy: 66.20%
Epoch [37/50], Step [100/391], Loss: 0.5803, Acc: 81.69%
Epoch [37/50], Step [200/391], Loss: 0.6010, Acc: 81.32%
Epoch [37/50], Step [300/391], Loss: 0.6078, Acc: 81.23%
Epoch [37/50] Test Accuracy: 66.55%
Epoch [38/50], Step [100/391], Loss: 0.5643, Acc: 82.35%
Epoch [38/50], Step [200/391], Loss: 0.5767, Acc: 81.98%
Epoch [38/50], Step [300/391], Loss: 0.5794, Acc: 81.89%
Epoch [38/50] Test Accuracy: 67.02%
Epoch [39/50], Step [100/391], Loss: 0.5335, Acc: 83.33%
Epoch [39/50], Step [200/391], Loss: 0.5534, Acc: 82.98%
Epoch [39/50], Step [300/391], Loss: 0.5569, Acc: 82.75%
Epoch [39/50] Test Accuracy: 66.56%
Epoch [40/50], Step [100/391], Loss: 0.5117, Acc: 84.20%
Epoch [40/50], Step [200/391], Loss: 0.5364, Acc: 83.65%
Epoch [40/50], Step [300/391], Loss: 0.5423, Acc: 83.37%
Epoch [40/50] Test Accuracy: 66.80%
Epoch [41/50], Step [100/391], Loss: 0.5065, Acc: 83.89%
Epoch [41/50], Step [200/391], Loss: 0.5056, Acc: 84.06%
Epoch [41/50], Step [300/391], Loss: 0.5186, Acc: 83.88%
Epoch [41/50] Test Accuracy: 67.21%
Epoch [42/50], Step [100/391], Loss: 0.4901, Acc: 84.59%
Epoch [42/50], Step [200/391], Loss: 0.5113, Acc: 84.39%
Epoch [42/50], Step [300/391], Loss: 0.4966, Acc: 84.21%
Epoch [42/50] Test Accuracy: 67.69%
```

```
Epoch [43/50], Step [100/391], Loss: 0.4786, Acc: 84.91%
Epoch [43/50], Step [200/391], Loss: 0.4788, Acc: 84.94%
Epoch [43/50], Step [300/391], Loss: 0.4739, Acc: 84.98%
Epoch [43/50] Test Accuracy: 67.46%
Epoch [44/50], Step [100/391], Loss: 0.4769, Acc: 84.97%
Epoch [44/50], Step [200/391], Loss: 0.4731, Acc: 85.10%
Epoch [44/50], Step [300/391], Loss: 0.4789, Acc: 84.99%
Epoch [44/50] Test Accuracy: 67.56%
Epoch [45/50], Step [100/391], Loss: 0.4615, Acc: 85.38%
Epoch [45/50], Step [200/391], Loss: 0.4577, Acc: 85.53%
Epoch [45/50], Step [300/391], Loss: 0.4578, Acc: 85.41%
Epoch [45/50] Test Accuracy: 67.97%
Epoch [46/50], Step [100/391], Loss: 0.4573, Acc: 85.70%
Epoch [46/50], Step [200/391], Loss: 0.4455, Acc: 85.91%
Epoch [46/50], Step [300/391], Loss: 0.4408, Acc: 85.95%
Epoch [46/50] Test Accuracy: 67.94%
Epoch [47/50], Step [100/391], Loss: 0.4510, Acc: 85.77%
Epoch [47/50], Step [200/391], Loss: 0.4275, Acc: 86.13%
Epoch [47/50], Step [300/391], Loss: 0.4393, Acc: 86.21%
Epoch [47/50] Test Accuracy: 68.18%
Epoch [48/50], Step [100/391], Loss: 0.4336, Acc: 86.59%
Epoch [48/50], Step [200/391], Loss: 0.4303, Acc: 86.45%
Epoch [48/50], Step [300/391], Loss: 0.4502, Acc: 86.28%
Epoch [48/50] Test Accuracy: 68.05%
Epoch [49/50], Step [100/391], Loss: 0.4305, Acc: 86.44%
Epoch [49/50], Step [200/391], Loss: 0.4329, Acc: 86.57%
Epoch [49/50], Step [300/391], Loss: 0.4356, Acc: 86.45%
Epoch [49/50] Test Accuracy: 68.13%
Epoch [50/50], Step [100/391], Loss: 0.4228, Acc: 86.88%
Epoch [50/50], Step [200/391], Loss: 0.4313, Acc: 86.68%
Epoch [50/50], Step [300/391], Loss: 0.4366, Acc: 86.47%
Epoch [50/50] Test Accuracy: 68.19%
Best Test Accuracy: 68.19%
```


Confusion Matrix
True
Predicted


Training Loss vs. Epoch
Training Loss
Loss
Epoch


Training and Test Accuracy vs. Epoch
Training Accuracy
Test Accuracy
Accuracy (%)
Epoch

[ ]:
```python
## VGG Net -> CIFAR-10

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,␣
 ↪precision_score
import seaborn as sns
import numpy as np


## NUM EPOCHS
num_epochs = 20

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# VGG16 for CIFAR-10 (only changing num_classes default)
class VGG16(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG16, self).__init__()
        # Block 1
        self.features = nn.Sequential(
            # Block 1 (64 channels)
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 2 (128 channels)
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
```

```
            # Block 3 (256 channels)
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 4 (512 channels)
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 5 (512 channels)
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Classifier
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(512 * 1 * 1, 4096),  # CIFAR-10 images are 32x32, after 5
↪max-pooling layers: 1x1
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes)
```

```
        )

        # Initialize weights
        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

# CIFAR-10 Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,␣
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
```

```python
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = VGG16(num_classes=10).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,␣
↪weight_decay=5e-4)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
 ↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
 ↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')

print(f'Best Test Accuracy: {best_acc:.2f}%')

# Replace the confusion matrix section with:
model.eval()
all_preds = []
```

```
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()

# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()

# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

[ ]:
```
## VGG Net ->CIFAR-100

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,␣
 ↪precision_score
import seaborn as sns
import numpy as np

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

## NUM EPOCHS
num_epochs = 20

# VGG16 for CIFAR-100
class VGG16(nn.Module):
    def __init__(self, num_classes=100):
        super(VGG16, self).__init__()
        # Block 1
        self.features = nn.Sequential(
            # Block 1 (64 channels)
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 2 (128 channels)
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 3 (256 channels)
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 4 (512 channels)
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 5 (512 channels)
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Classifier
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(512 * 1 * 1, 4096),  # CIFAR-100 images are 32x32, after 5 max-pooling layers: 1x1
```

```
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes)
        )

        # Initialize weights
        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

# CIFAR-100 Dataset
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
```

```python
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,␣
↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR100(root='./data', train=False,␣
↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = VGG16(num_classes=100).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,␣
↪weight_decay=5e-4)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')
```

```python
print(f'Best Test Accuracy: {best_acc:.2f}%')

# Replace the confusion matrix section with:
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()

# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()
```

```
# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

[ ]:
```
## ResNet11 -> CIFAR-10

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,␣
↪precision_score
import seaborn as sns
import numpy as np


## NUM EPOCHS
num_epochs = 20

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# Basic ResNet block
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection
```

```python
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += self.shortcut(identity)
        out = self.relu(out)

        return out

# ResNet11
class ResNet11(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet11, self).__init__()

        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # ResNet blocks
        self.layer1 = BasicBlock(64, 64)
        self.layer2 = BasicBlock(64, 128, stride=2)
        self.layer3 = BasicBlock(128, 256, stride=2)
        self.layer4 = BasicBlock(256, 512, stride=2)

        # Average pooling and classifier
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

        # Initialize weights
        self._initialize_weights()
```

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

# CIFAR-10 Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,␣
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet11(num_classes=10).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,␣
 ↪weight_decay=5e-4)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')
```

```python
print(f'Best Test Accuracy: {best_acc:.2f}%')

# Replace the confusion matrix section with:
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()

# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()
```

```
# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

[ ]:
```
## ResNet11 -> CIFAR-100

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,␣
↪precision_score
import seaborn as sns
import numpy as np


## NUM EPOCHS
num_epochs = 20

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# Basic ResNet block
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection
```

```
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += self.shortcut(identity)
        out = self.relu(out)

        return out

# ResNet11
class ResNet11(nn.Module):
    def __init__(self, num_classes=100):
        super(ResNet11, self).__init__()

        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # ResNet blocks
        self.layer1 = BasicBlock(64, 64)
        self.layer2 = BasicBlock(64, 128, stride=2)
        self.layer3 = BasicBlock(128, 256, stride=2)
        self.layer4 = BasicBlock(256, 512, stride=2)

        # Average pooling and classifier
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

        # Initialize weights
        self._initialize_weights()
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

# CIFAR-100 Dataset
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
```

```python
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet11(num_classes=100).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')
```

```python
print(f'Best Test Accuracy: {best_acc:.2f}%')

# Replace the confusion matrix section with:
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()

# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()
```

```
# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

[ ]:
```
## ResNet18 -> CIFAR-10

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,␣
↪precision_score
import seaborn as sns
import numpy as np


## NUM EPOCHS
num_epochs = 20

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# Basic ResNet block
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection
```

```
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,␣
 ↪stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += self.shortcut(identity)
        out = self.relu(out)

        return out

# ResNet18
class ResNet18(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet18, self).__init__()

        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # ResNet layers
        self.layer1 = nn.Sequential(
            BasicBlock(64, 64),
            BasicBlock(64, 64)
        )
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, stride=2),
            BasicBlock(128, 128)
        )
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, stride=2),
            BasicBlock(256, 256)
        )
```

```
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, stride=2),
            BasicBlock(512, 512)
        )

        # Average pooling and classifier
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

        # Initialize weights
        self._initialize_weights()

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

# CIFAR-10 Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,␣
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet18(num_classes=10).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,␣
 ↪weight_decay=5e-4)

# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
```

```
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()

    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
```

```
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')

print(f'Best Test Accuracy: {best_acc:.2f}%')

# Replace the confusion matrix section with:
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()
```

```
# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()

# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```

[ ]:
```
## ResNet11 -> CIFAR-100

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import warnings
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, f1_score, recall_score,
 ↪precision_score
import seaborn as sns
import numpy as np


## NUM EPOCHS
num_epochs = 20

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")

# Basic ResNet block
class BasicBlock(nn.Module):
```

```
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut connection
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,␣
↪stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += self.shortcut(identity)
        out = self.relu(out)

        return out

# ResNet18
class ResNet18(nn.Module):
    def __init__(self, num_classes=100):
        super(ResNet18, self).__init__()

        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # ResNet layers
        self.layer1 = nn.Sequential(
```

```python
            BasicBlock(64, 64),
            BasicBlock(64, 64)
        )
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, stride=2),
            BasicBlock(128, 128)
        )
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, stride=2),
            BasicBlock(256, 256)
        )
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, stride=2),
            BasicBlock(512, 512)
        )

        # Average pooling and classifier
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

        # Initialize weights
        self._initialize_weights()

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
 ↪nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
```

```python
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# Data transforms for training and testing
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

# CIFAR-100 Dataset
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,␣
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR100(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=2)

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet18(num_classes=100).to(device)

# After model definition but before training loop
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total Parameters: {total_params:,}')
print(f'Trainable Parameters: {trainable_params:,}')

# Training loop


# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,␣
 ↪weight_decay=5e-4)
```

```
# Learning rate scheduler
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

# lists to store metrics  -  initiate beefore training loop
train_losses = []
train_accs = []
test_accs = []

best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        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()
        epoch_loss += loss.item()  # Accumulate loss for the entire epoch
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(trainloader)}], '
                  f'Loss: {running_loss/100:.4f}, Acc: {100*correct/total:.
↪2f}%')
            running_loss = 0.0

    # Calculate and store epoch metrics
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)

    # Adjust learning rate
    scheduler.step()
```

```
    # Evaluate on test set after each epoch
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()

    test_acc = 100 * test_correct / test_total
    test_accs.append(test_acc)  # Store test accuracy
    print(f'Epoch [{epoch+1}/{num_epochs}] Test Accuracy: {test_acc:.2f}%')

    # Save best model
    if test_acc > best_acc:
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')

print(f'Best Test Accuracy: {best_acc:.2f}%')


model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Calculate metrics
f1 = f1_score(all_labels, all_preds, average='macro')
recall = recall_score(all_labels, all_preds, average='macro')
precision = precision_score(all_labels, all_preds, average='macro')

print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')
```

```python
# Create confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()

# Plot the curves
plt.figure(figsize=(12, 4))

# Loss curve
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.legend()

# Accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_accs, label='Training Accuracy')
plt.plot(test_accs, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()
```