Problem 1 (30 pts)

AlexNet is originally proposed for 227*227 image sizes. It may be too complex for the CIFAR-10 and CIFAR-100 datasets, in particular, due to the low resolution of the initial images; try simplifying the model to make the training faster while ensuring that the accuracy stays relatively high. Report the training loss, validation loss, and validation accuracy. Also, report the number of parameters in your modified version of AlexNet and compare it against the number of parameters in the original AlexNet architectures. Here is a good reference guide to AlexNet: https://www.kaggle.com/code/blurredmachine/alexnet-architecture-a-complete-guide

Links to an external site.

Explore the option of applying Dropout techniques for training your customized AlexNet. Compare the training and validation results against the baseline model without any dropout. Also, compare the results between CIFAR-10 and CIFAR-100.

Here are some information about CIFAR-100

https://paperswithcode.com/dataset/cifar-100

Links to an external site.

https://pytorch.org/vision/main/generated/torchvision.datasets.CIFAR100.html

In [4]:
```
# 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, t
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu
```

```
# 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)}]
            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
```

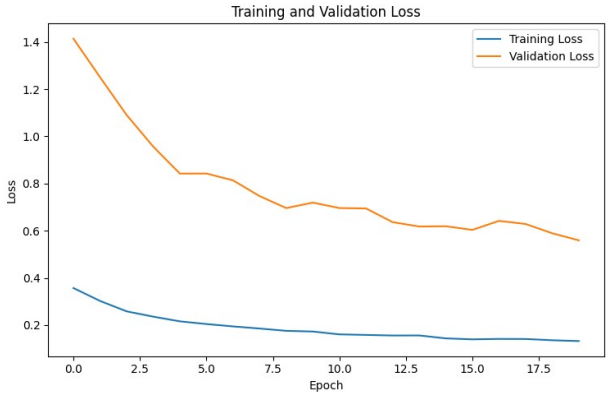
```
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (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')

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

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


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

# plotting loss curves
plt.subplot(1, 2, 1)
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()

# plotting 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()

# Plot confusion matrix
model.eval()
```

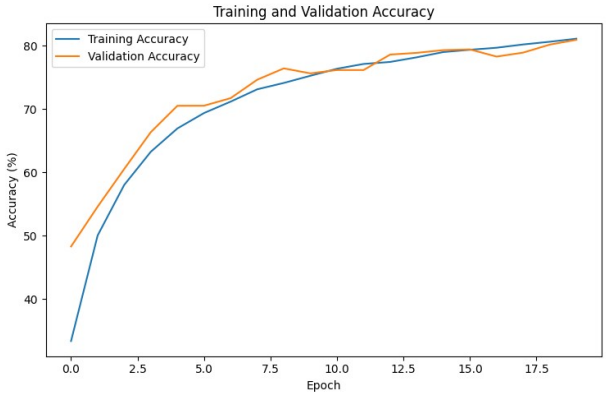
```
all_preds = []
all_labels = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        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()
# Calculate precision, recall, f1-score for each class
from sklearn.metrics import precision_recall_fscore_support, accuracy_score
precision, recall, f1, _ = precision_recall_fscore_support(all_labels, all_preds, a
accuracy = accuracy_score(all_labels, all_preds)
print(f"\nOverall Accuracy: {accuracy*100:.2f}%\n")
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{f
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}%")
```

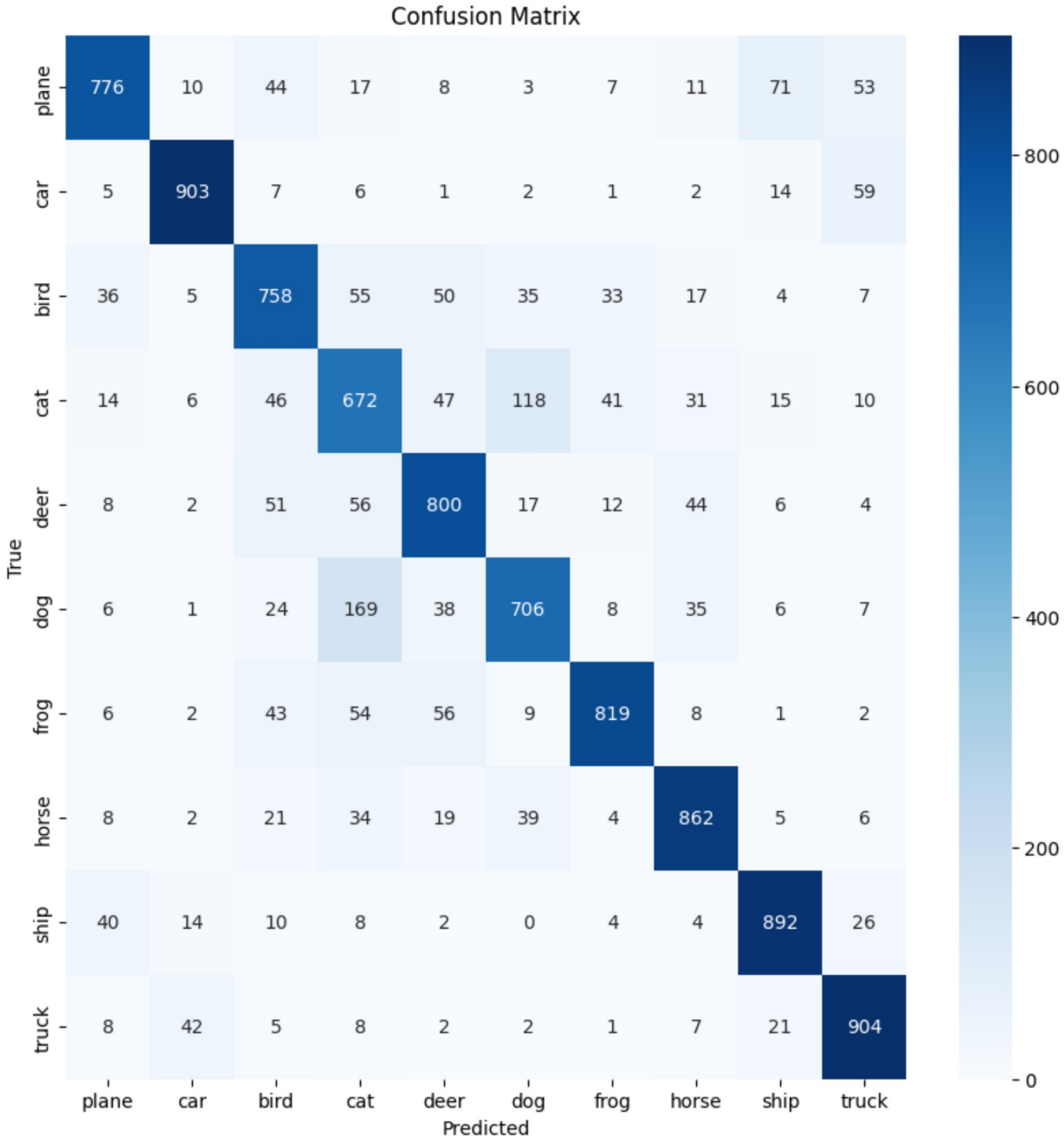
```
Epoch [1/20], Step [100/391], Loss: 2.0382
Epoch [1/20], Step [200/391], Loss: 1.7828
Epoch [1/20], Step [300/391], Loss: 1.6285
Epoch [1/20]:
Train Loss: 0.3568, Train Acc: 33.38%
Test Loss: 1.4139, Test Acc: 48.30%
Epoch [2/20], Step [100/391], Loss: 1.4378
Epoch [2/20], Step [200/391], Loss: 1.3744
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
```






In [1]:
```
#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
```

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

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

# Data transforms
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,
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n
testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True,
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AlexNet(num_classes=100).to(device)

# Training loop
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)
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

    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()
        _, 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)
    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()

# Plot the curves
```

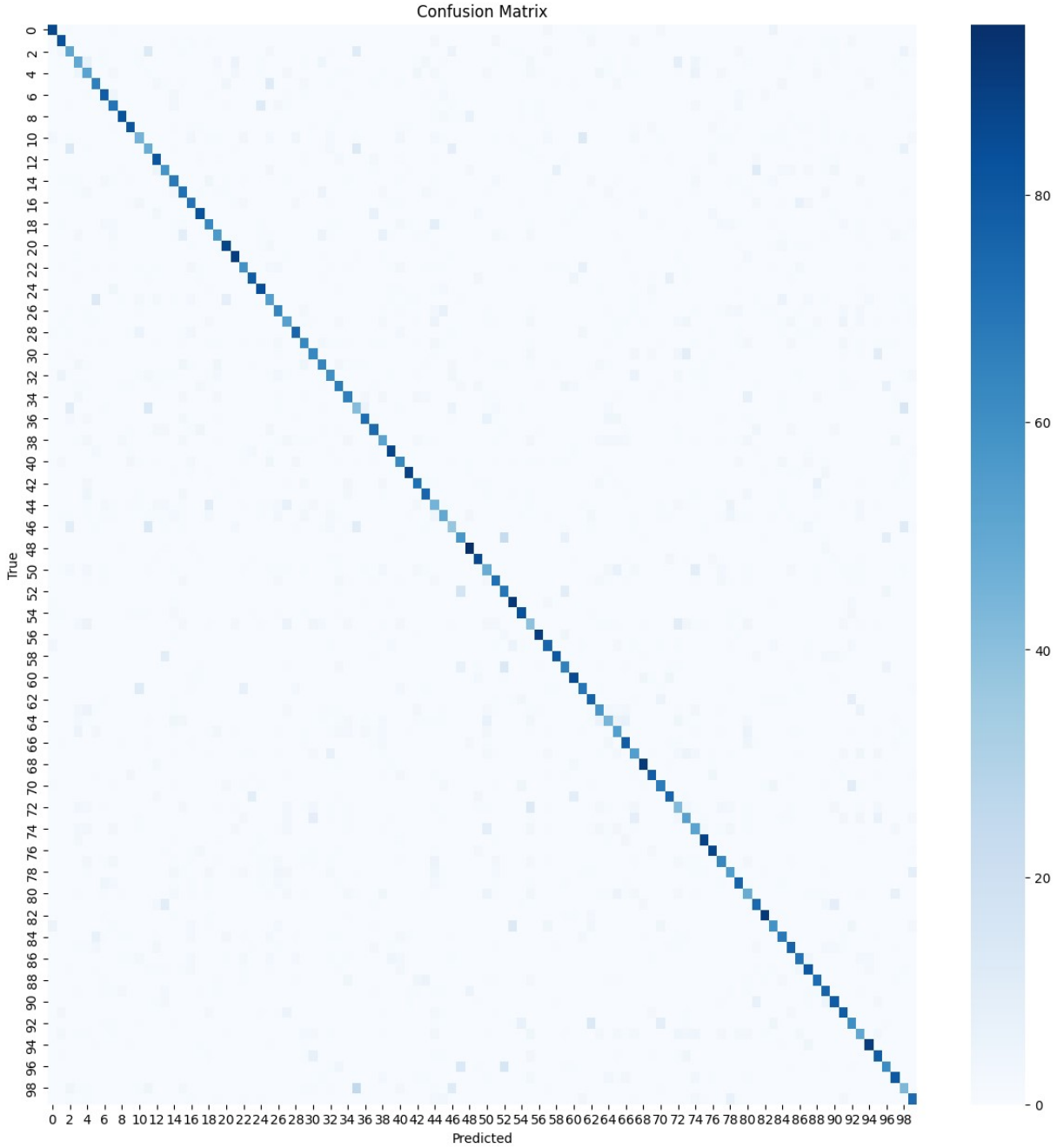
```
plt.figure(figsize=(12, 4))

# 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()
```

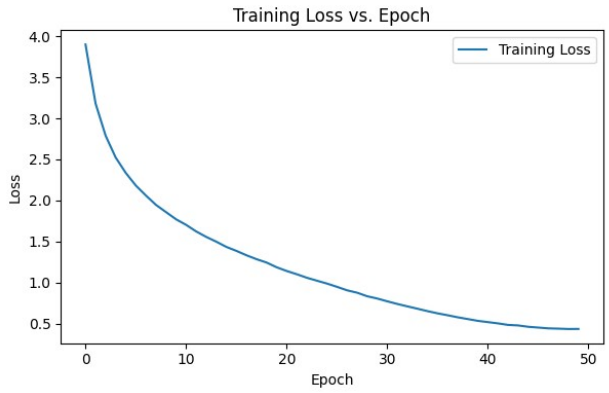
```
Epoch [1/50], Step [100/391], Loss: 4.3987, Acc: 3.55%
Epoch [1/50], Step [200/391], Loss: 3.9365, Acc: 6.06%
Epoch [1/50], Step [300/391], Loss: 3.6848, Acc: 8.01%
Epoch [1/50] Test Accuracy: 17.03%
Epoch [2/50], Step [100/391], Loss: 3.3500, Acc: 18.34%
Epoch [2/50], Step [200/391], Loss: 3.2556, 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%
```

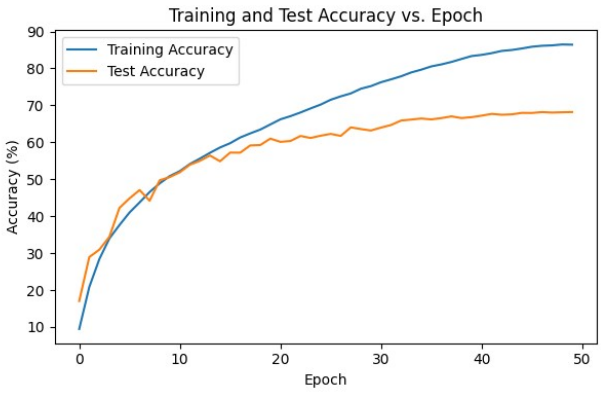
```
Epoch [15/50], Step [100/391], Loss: 1.4153, Acc: 59.14%
Epoch [15/50], Step [200/391], Loss: 1.4281, Acc: 58.98%
Epoch [15/50], Step [300/391], Loss: 1.4237, Acc: 58.97%
Epoch [15/50] Test Accuracy: 54.86%
Epoch [16/50], Step [100/391], Loss: 1.3751, Acc: 60.39%
Epoch [16/50], Step [200/391], Loss: 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%
```




Problem 2 (30 pts)

Repeat the problem 1 this time for VGGNet. Identify the VGG configuration that matches the

nearest number of parameters to the AlexNet Architecture that you did for problem 1 for CIFAR-10 and CIFAR-100 datasets. Compare your training and evaluation results against AlexNet in Problem 1.

Here is a good reference guide to AlexNet: https://www.kaggle.com/code/blurredmachine/vggnet-16-architecture-a-complete-guide Links to an external site.

Links to an external site.

Explore the option of applying Dropout techniques for training your customized AlexNet. Compare the training and validation results against the baseline model without any dropout. Also, compare the results between CIFAR-10 and CIFAR-100.

In [1]:
```
# 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_sco
import seaborn as sns
import numpy as np

num_epochs = 20

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

# VGG16 for CIFAR-10
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
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes)
        )

        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        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='rel
                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)
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, t
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu

# 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 = []
```

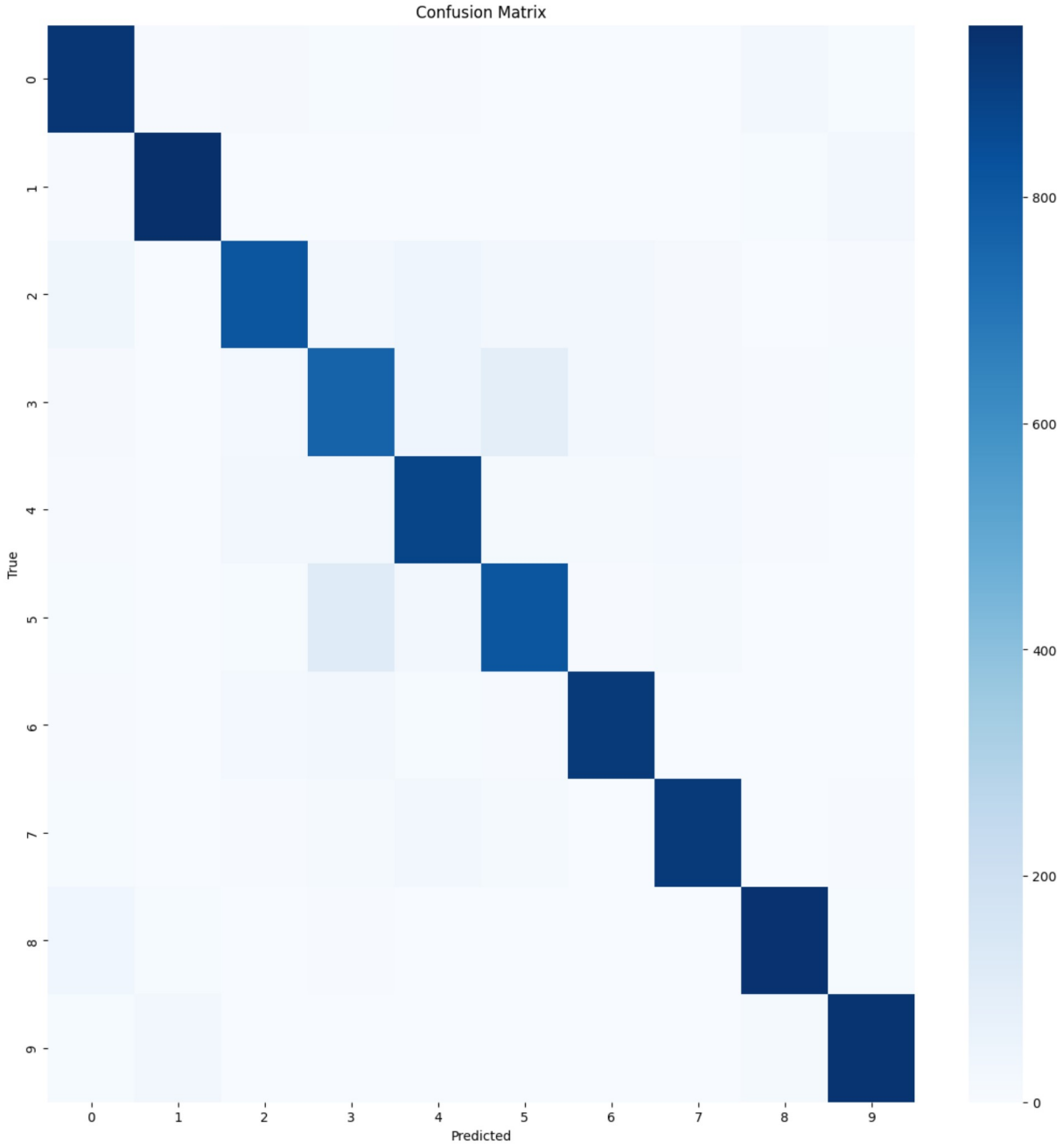
```
best_acc = 0.0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    epoch_loss = 0.0  # Track total loss for the epoch

    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:
```

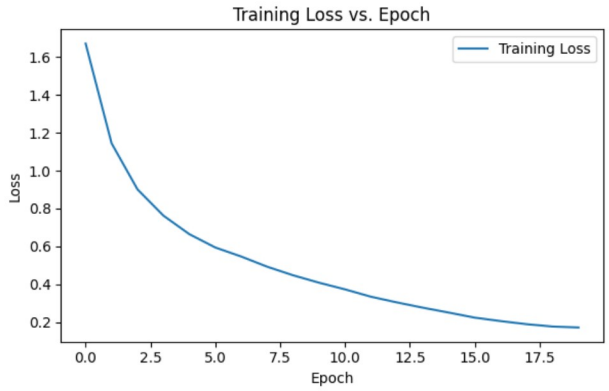
```
        best_acc = test_acc
        torch.save(model.state_dict(), 'best_model.pth')

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')
```

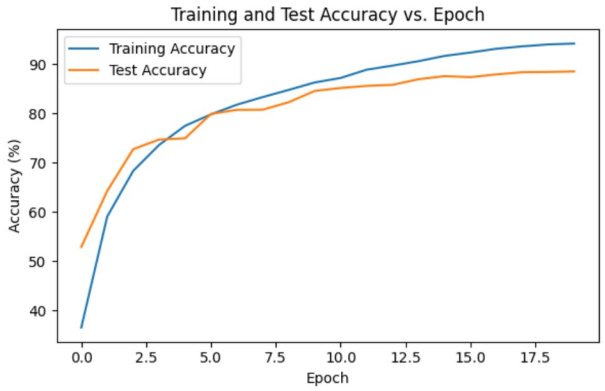
```
plt.ylabel('Accuracy (%)')
plt.title('Training and Test Accuracy vs. Epoch')
plt.legend()

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

```
Files already downloaded and verified
Files already downloaded and verified
Total Parameters: 33,646,666
Trainable Parameters: 33,646,666
Epoch [1/20], Step [100/391], Loss: 2.0779, Acc: 19.50%
Epoch [1/20], Step [200/391], Loss: 1.6966, Acc: 26.93%
Epoch [1/20], Step [300/391], Loss: 1.5091, Acc: 32.58%
Epoch [1/20] Test Accuracy: 52.88%
Epoch [2/20], Step [100/391], Loss: 1.2411, Acc: 54.93%
Epoch [2/20], Step [200/391], Loss: 1.1934, Acc: 56.38%
Epoch [2/20], Step [300/391], Loss: 1.0974, Acc: 57.69%
Epoch [2/20] Test Accuracy: 64.24%
Epoch [3/20], Step [100/391], Loss: 0.9544, Acc: 66.29%
Epoch [3/20], Step [200/391], Loss: 0.9109, Acc: 67.09%
Epoch [3/20], Step [300/391], Loss: 0.8765, Acc: 67.76%
Epoch [3/20] Test Accuracy: 72.72%
Epoch [4/20], Step [100/391], Loss: 0.7909, Acc: 72.55%
Epoch [4/20], Step [200/391], Loss: 0.7753, Acc: 72.75%
Epoch [4/20], Step [300/391], Loss: 0.7317, Acc: 73.47%
Epoch [4/20] Test Accuracy: 74.68%
Epoch [5/20], Step [100/391], Loss: 0.6762, Acc: 77.04%
Epoch [5/20], Step [200/391], Loss: 0.6772, Acc: 76.96%
Epoch [5/20], Step [300/391], Loss: 0.6582, Acc: 77.24%
Epoch [5/20] Test Accuracy: 74.93%
Epoch [6/20], Step [100/391], Loss: 0.5972, Acc: 79.74%
Epoch [6/20], Step [200/391], Loss: 0.6031, Acc: 79.65%
Epoch [6/20], Step [300/391], Loss: 0.5883, Acc: 79.75%
Epoch [6/20] Test Accuracy: 79.87%
Epoch [7/20], Step [100/391], Loss: 0.5529, Acc: 81.52%
Epoch [7/20], Step [200/391], Loss: 0.5531, Acc: 81.67%
Epoch [7/20], Step [300/391], Loss: 0.5433, Acc: 81.76%
Epoch [7/20] Test Accuracy: 80.71%
Epoch [8/20], Step [100/391], Loss: 0.5019, Acc: 82.88%
Epoch [8/20], Step [200/391], Loss: 0.4993, Acc: 83.08%
Epoch [8/20], Step [300/391], Loss: 0.4888, Acc: 83.21%
Epoch [8/20] Test Accuracy: 80.74%
Epoch [9/20], Step [100/391], Loss: 0.4541, Acc: 84.32%
Epoch [9/20], Step [200/391], Loss: 0.4463, Acc: 84.60%
Epoch [9/20], Step [300/391], Loss: 0.4438, Acc: 84.79%
Epoch [9/20] Test Accuracy: 82.26%
Epoch [10/20], Step [100/391], Loss: 0.4095, Acc: 86.11%
Epoch [10/20], Step [200/391], Loss: 0.4114, Acc: 86.12%
Epoch [10/20], Step [300/391], Loss: 0.4040, Acc: 86.20%
Epoch [10/20] Test Accuracy: 84.55%
Epoch [11/20], Step [100/391], Loss: 0.3617, Acc: 87.63%
Epoch [11/20], Step [200/391], Loss: 0.3754, Acc: 87.38%
Epoch [11/20], Step [300/391], Loss: 0.3782, Acc: 87.22%
Epoch [11/20] Test Accuracy: 85.14%
Epoch [12/20], Step [100/391], Loss: 0.3315, Acc: 88.89%
Epoch [12/20], Step [200/391], Loss: 0.3351, Acc: 88.79%
Epoch [12/20], Step [300/391], Loss: 0.3367, Acc: 88.80%
Epoch [12/20] Test Accuracy: 85.57%
Epoch [13/20], Step [100/391], Loss: 0.2918, Acc: 90.09%
Epoch [13/20], Step [200/391], Loss: 0.3012, Acc: 89.84%
Epoch [13/20], Step [300/391], Loss: 0.3130, Acc: 89.70%
Epoch [13/20] Test Accuracy: 85.78%
```

```
Epoch [14/20], Step [100/391], Loss: 0.2765, Acc: 90.48%
Epoch [14/20], Step [200/391], Loss: 0.2827, Acc: 90.38%
Epoch [14/20], Step [300/391], Loss: 0.2722, Acc: 90.53%
Epoch [14/20] Test Accuracy: 86.92%
Epoch [15/20], Step [100/391], Loss: 0.2510, Acc: 91.56%
Epoch [15/20], Step [200/391], Loss: 0.2524, Acc: 91.61%
Epoch [15/20], Step [300/391], Loss: 0.2506, Acc: 91.58%
Epoch [15/20] Test Accuracy: 87.54%
Epoch [16/20], Step [100/391], Loss: 0.2282, Acc: 92.09%
Epoch [16/20], Step [200/391], Loss: 0.2341, Acc: 91.97%
Epoch [16/20], Step [300/391], Loss: 0.2228, Acc: 92.16%
Epoch [16/20] Test Accuracy: 87.34%
Epoch [17/20], Step [100/391], Loss: 0.2058, Acc: 93.01%
Epoch [17/20], Step [200/391], Loss: 0.1994, Acc: 93.21%
Epoch [17/20], Step [300/391], Loss: 0.2125, Acc: 93.11%
Epoch [17/20] Test Accuracy: 87.90%
Epoch [18/20], Step [100/391], Loss: 0.1858, Acc: 93.86%
Epoch [18/20], Step [200/391], Loss: 0.1884, Acc: 93.72%
Epoch [18/20], Step [300/391], Loss: 0.1844, Acc: 93.71%
Epoch [18/20] Test Accuracy: 88.34%
Epoch [19/20], Step [100/391], Loss: 0.1803, Acc: 93.97%
Epoch [19/20], Step [200/391], Loss: 0.1781, Acc: 94.00%
Epoch [19/20], Step [300/391], Loss: 0.1774, Acc: 93.94%
Epoch [19/20] Test Accuracy: 88.39%
Epoch [20/20], Step [100/391], Loss: 0.1686, Acc: 94.20%
Epoch [20/20], Step [200/391], Loss: 0.1697, Acc: 94.21%
Epoch [20/20], Step [300/391], Loss: 0.1783, Acc: 94.17%
Epoch [20/20] Test Accuracy: 88.50%
Best Test Accuracy: 88.50%

Model Performance Metrics:
F1 Score (macro): 0.8847
Recall (macro): 0.8850
Precision (macro): 0.8849
```




In [2]:
```
# 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_sco
import seaborn as sns
import numpy as np

warnings.filterwarnings("ignore", message="TypedStorage is deprecated")
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 ma
            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='rel
                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,
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True,
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu

# 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
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
```

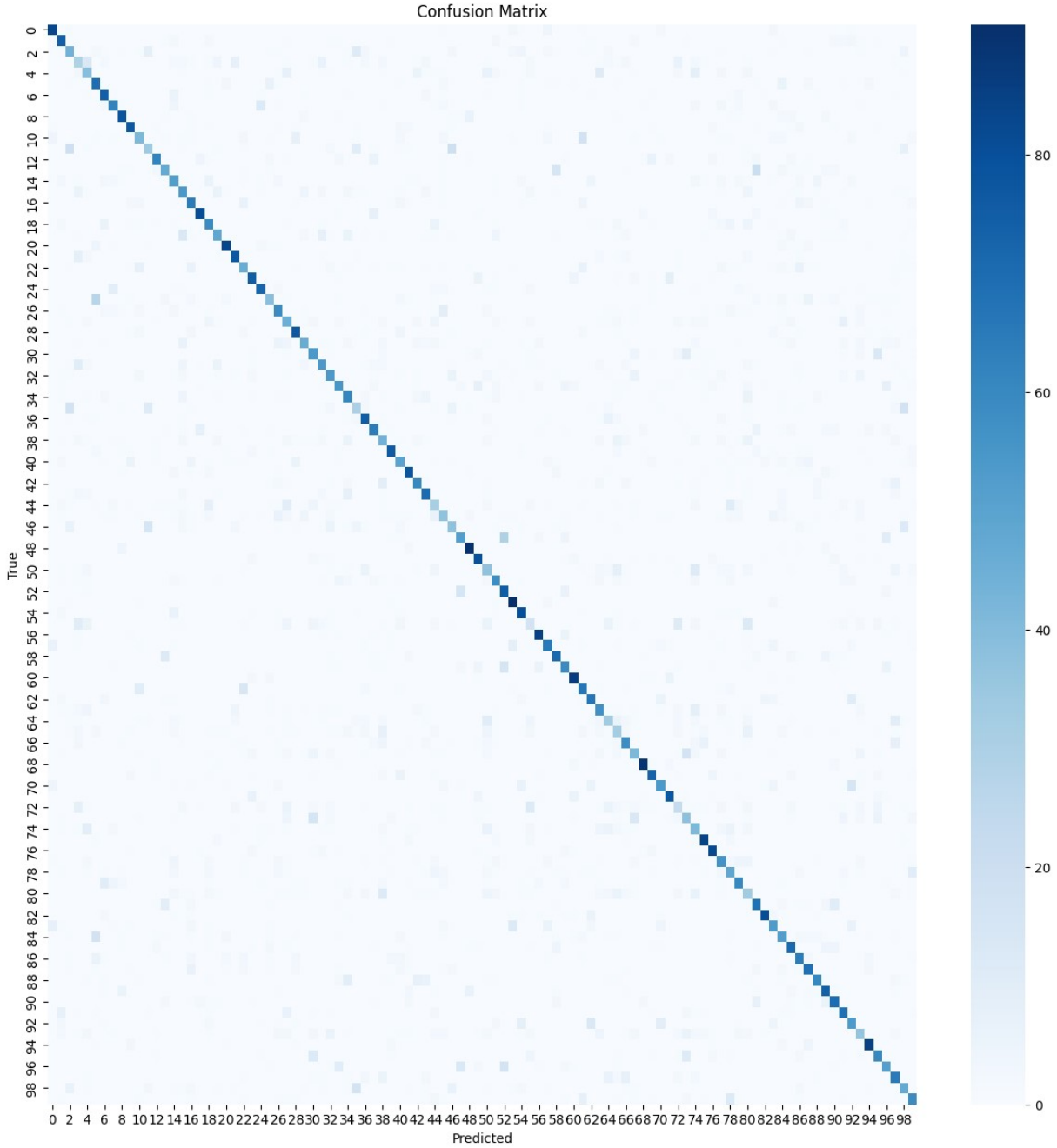
```
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        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:')
```

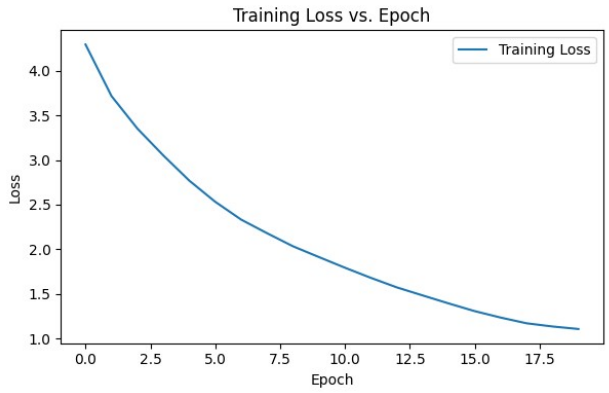
```
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# 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()
```

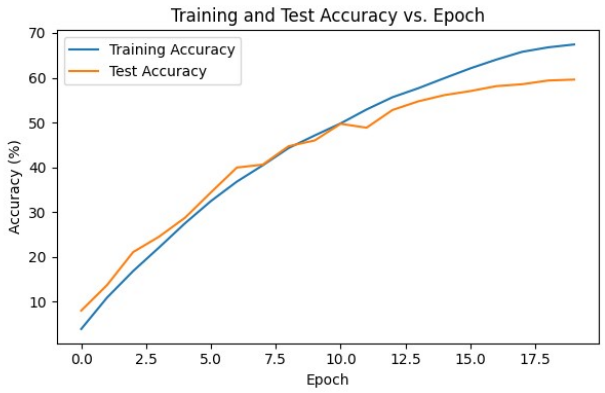
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz to ./data\cifa
r-100-python.tar.gz
100.0%
```

```
Extracting ./data\cifar-100-python.tar.gz to ./data
Files already downloaded and verified
Total Parameters: 34,015,396
Trainable Parameters: 34,015,396
Epoch [1/20], Step [100/391], Loss: 4.5517, Acc: 1.58%
Epoch [1/20], Step [200/391], Loss: 4.3901, Acc: 2.17%
Epoch [1/20], Step [300/391], Loss: 4.1905, Acc: 3.07%
Epoch [1/20] Test Accuracy: 7.99%
Epoch [2/20], Step [100/391], Loss: 3.8684, Acc: 8.73%
Epoch [2/20], Step [200/391], Loss: 3.7471, Acc: 9.30%
Epoch [2/20], Step [300/391], Loss: 3.6758, Acc: 10.21%
Epoch [2/20] Test Accuracy: 13.72%
Epoch [3/20], Step [100/391], Loss: 3.4510, Acc: 15.26%
Epoch [3/20], Step [200/391], Loss: 3.4090, Acc: 15.54%
Epoch [3/20], Step [300/391], Loss: 3.3117, Acc: 16.16%
Epoch [3/20] Test Accuracy: 21.06%
Epoch [4/20], Step [100/391], Loss: 3.1266, Acc: 20.76%
Epoch [4/20], Step [200/391], Loss: 3.1153, Acc: 20.70%
Epoch [4/20], Step [300/391], Loss: 3.0024, Acc: 21.64%
Epoch [4/20] Test Accuracy: 24.50%
Epoch [5/20], Step [100/391], Loss: 2.8334, Acc: 25.96%
Epoch [5/20], Step [200/391], Loss: 2.8004, Acc: 26.27%
Epoch [5/20], Step [300/391], Loss: 2.7392, Acc: 26.96%
Epoch [5/20] Test Accuracy: 28.73%
Epoch [6/20], Step [100/391], Loss: 2.6048, Acc: 30.46%
Epoch [6/20], Step [200/391], Loss: 2.5508, Acc: 31.52%
Epoch [6/20], Step [300/391], Loss: 2.5076, Acc: 31.89%
Epoch [6/20] Test Accuracy: 34.39%
Epoch [7/20], Step [100/391], Loss: 2.3573, Acc: 36.24%
Epoch [7/20], Step [200/391], Loss: 2.3571, Acc: 36.16%
Epoch [7/20], Step [300/391], Loss: 2.3130, Acc: 36.49%
Epoch [7/20] Test Accuracy: 39.94%
Epoch [8/20], Step [100/391], Loss: 2.1983, Acc: 40.07%
Epoch [8/20], Step [200/391], Loss: 2.1851, Acc: 40.21%
Epoch [8/20], Step [300/391], Loss: 2.1846, Acc: 40.20%
Epoch [8/20] Test Accuracy: 40.60%
Epoch [9/20], Step [100/391], Loss: 2.0498, Acc: 43.72%
Epoch [9/20], Step [200/391], Loss: 2.0501, Acc: 43.73%
Epoch [9/20], Step [300/391], Loss: 2.0267, Acc: 43.99%
Epoch [9/20] Test Accuracy: 44.73%
Epoch [10/20], Step [100/391], Loss: 1.9137, Acc: 46.66%
Epoch [10/20], Step [200/391], Loss: 1.9248, Acc: 46.92%
Epoch [10/20], Step [300/391], Loss: 1.9071, Acc: 46.91%
Epoch [10/20] Test Accuracy: 45.99%
Epoch [11/20], Step [100/391], Loss: 1.7880, Acc: 50.00%
Epoch [11/20], Step [200/391], Loss: 1.7868, Acc: 49.98%
Epoch [11/20], Step [300/391], Loss: 1.7830, Acc: 49.99%
Epoch [11/20] Test Accuracy: 49.71%
Epoch [12/20], Step [100/391], Loss: 1.6744, Acc: 52.95%
Epoch [12/20], Step [200/391], Loss: 1.6952, Acc: 52.97%
Epoch [12/20], Step [300/391], Loss: 1.6491, Acc: 53.11%
Epoch [12/20] Test Accuracy: 48.84%
Epoch [13/20], Step [100/391], Loss: 1.5679, Acc: 55.19%
Epoch [13/20], Step [200/391], Loss: 1.5653, Acc: 55.53%
Epoch [13/20], Step [300/391], Loss: 1.5810, Acc: 55.59%
Epoch [13/20] Test Accuracy: 52.80%
```

```
Epoch [14/20], Step [100/391], Loss: 1.4850, Acc: 57.52%
Epoch [14/20], Step [200/391], Loss: 1.4665, Acc: 57.79%
Epoch [14/20], Step [300/391], Loss: 1.4916, Acc: 57.83%
Epoch [14/20] Test Accuracy: 54.74%
Epoch [15/20], Step [100/391], Loss: 1.3937, Acc: 59.67%
Epoch [15/20], Step [200/391], Loss: 1.3692, Acc: 59.99%
Epoch [15/20], Step [300/391], Loss: 1.4150, Acc: 59.87%
Epoch [15/20] Test Accuracy: 56.12%
Epoch [16/20], Step [100/391], Loss: 1.2919, Acc: 62.64%
Epoch [16/20], Step [200/391], Loss: 1.3121, Acc: 62.42%
Epoch [16/20], Step [300/391], Loss: 1.3039, Acc: 62.32%
Epoch [16/20] Test Accuracy: 57.02%
Epoch [17/20], Step [100/391], Loss: 1.2153, Acc: 64.55%
Epoch [17/20], Step [200/391], Loss: 1.2323, Acc: 64.23%
Epoch [17/20], Step [300/391], Loss: 1.2440, Acc: 63.99%
Epoch [17/20] Test Accuracy: 58.12%
Epoch [18/20], Step [100/391], Loss: 1.1629, Acc: 66.02%
Epoch [18/20], Step [200/391], Loss: 1.1893, Acc: 65.72%
Epoch [18/20], Step [300/391], Loss: 1.1587, Acc: 65.69%
Epoch [18/20] Test Accuracy: 58.55%
Epoch [19/20], Step [100/391], Loss: 1.1261, Acc: 67.12%
Epoch [19/20], Step [200/391], Loss: 1.1367, Acc: 66.75%
Epoch [19/20], Step [300/391], Loss: 1.1431, Acc: 66.66%
Epoch [19/20] Test Accuracy: 59.38%
Epoch [20/20], Step [100/391], Loss: 1.1017, Acc: 67.49%
Epoch [20/20], Step [200/391], Loss: 1.1138, Acc: 67.44%
Epoch [20/20], Step [300/391], Loss: 1.1115, Acc: 67.38%
Epoch [20/20] Test Accuracy: 59.59%
Best Test Accuracy: 59.59%

Model Performance Metrics:
F1 Score (macro): 0.5920
Recall (macro): 0.5959
Precision (macro): 0.5929
```




## Problem 3 (40 pts)

The baseline model we did in lectures is called ResNet-11. Build a new version of ResNet

(ResNet-18). Train it on CIFAR-10 and CIFAR-100 datasets. Plot the training loss, validation loss, and validation accuracy. Compare the classification accuracy, and model size across the two versions of ResNet (11, 18). How does the complexity grow as you increase the network depth?

You can find some references for ResNet 18 here:

https://www.kaggle.com/code/ivankunyankin/resnet18-from-scratch-using-pytorch

Links to an external site.

Explore the dropout option for the two networks and report your training results and validation accuracy. Also, compare the results between CIFAR-10 and CIFAR-100.

In [3]:
```
# 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_sco
import seaborn as sns
import numpy as np

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=str
        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,
        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,
                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=Fals
        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='rel
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
```

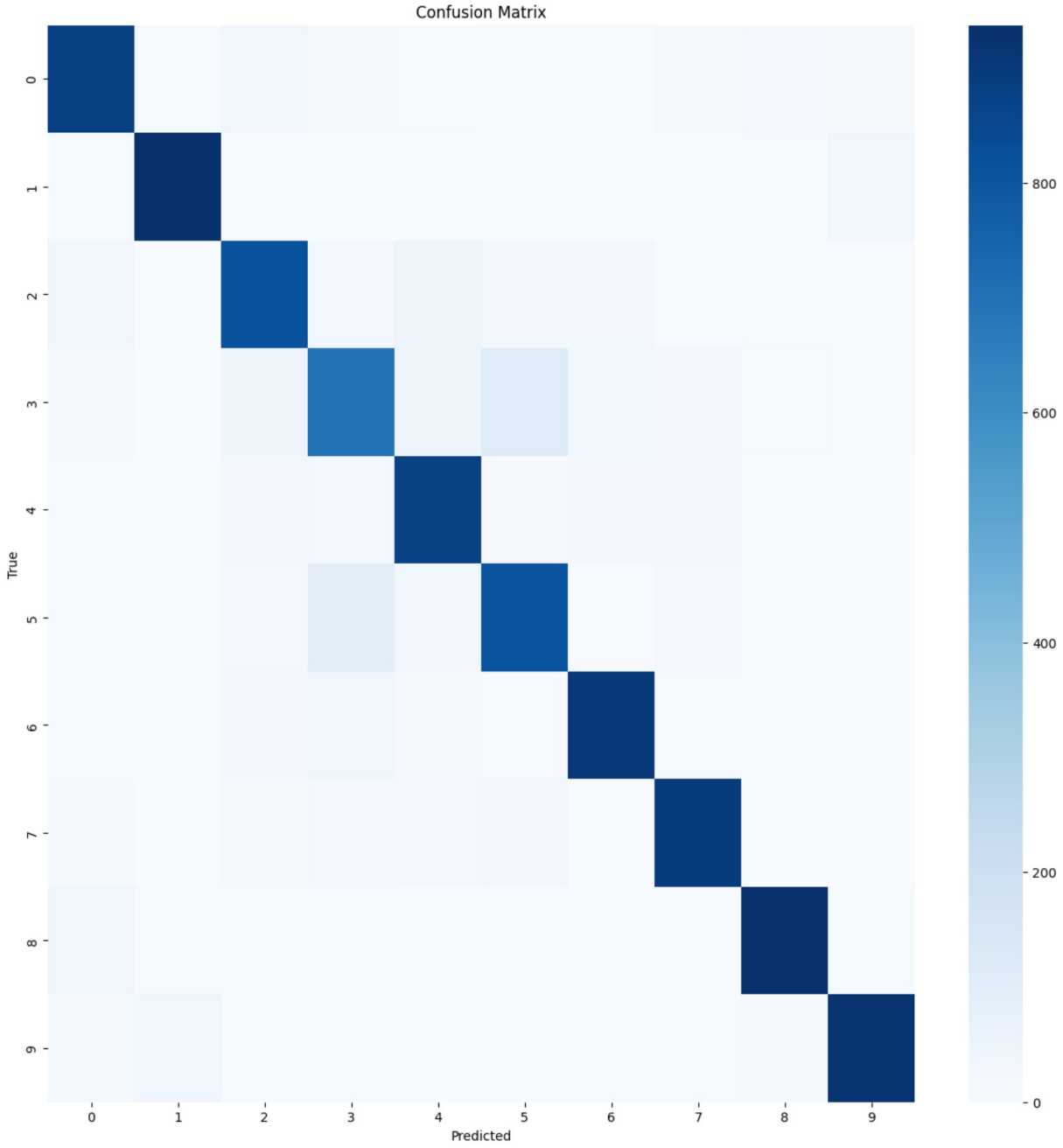
```
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

# 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, t
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet11(num_classes=10).to(device)
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
```

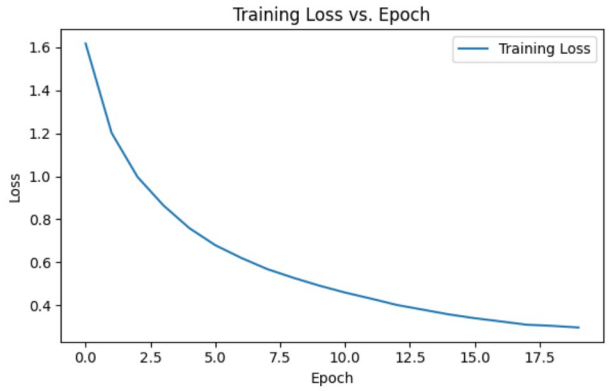
```
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        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

    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)
    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
```

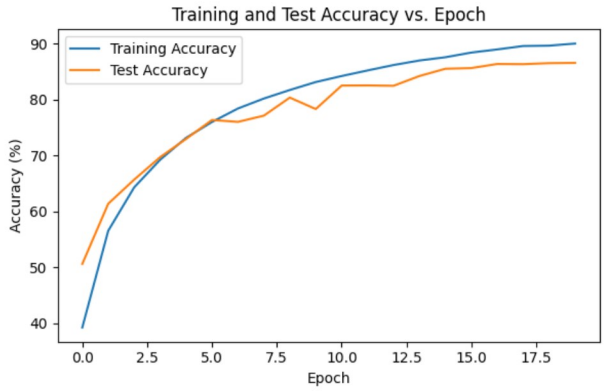
```
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# 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}')

# Plot confusion matrix
confmat = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(15, 15))
sns.heatmap(confmat, 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()
```

```
Files already downloaded and verified
Files already downloaded and verified
Total Parameters: 4,903,242
Trainable Parameters: 4,903,242
Epoch [1/20], Step [100/391], Loss: 1.9310, Acc: 26.66%
Epoch [1/20], Step [200/391], Loss: 1.6183, Acc: 32.38%
Epoch [1/20], Step [300/391], Loss: 1.4944, Acc: 36.53%
Epoch [1/20] Test Accuracy: 50.63%
Epoch [2/20], Step [100/391], Loss: 1.2966, Acc: 52.91%
Epoch [2/20], Step [200/391], Loss: 1.2239, Acc: 54.39%
Epoch [2/20], Step [300/391], Loss: 1.1534, Acc: 55.65%
Epoch [2/20] Test Accuracy: 61.39%
Epoch [3/20], Step [100/391], Loss: 1.0483, Acc: 62.79%
Epoch [3/20], Step [200/391], Loss: 1.0128, Acc: 63.07%
Epoch [3/20], Step [300/391], Loss: 0.9782, Acc: 63.65%
Epoch [3/20] Test Accuracy: 65.70%
Epoch [4/20], Step [100/391], Loss: 0.8886, Acc: 68.77%
Epoch [4/20], Step [200/391], Loss: 0.8793, Acc: 68.67%
Epoch [4/20], Step [300/391], Loss: 0.8601, Acc: 68.86%
Epoch [4/20] Test Accuracy: 69.72%
Epoch [5/20], Step [100/391], Loss: 0.7942, Acc: 71.88%
Epoch [5/20], Step [200/391], Loss: 0.7583, Acc: 72.52%
Epoch [5/20], Step [300/391], Loss: 0.7550, Acc: 72.84%
Epoch [5/20] Test Accuracy: 72.98%
Epoch [6/20], Step [100/391], Loss: 0.6881, Acc: 75.62%
Epoch [6/20], Step [200/391], Loss: 0.6715, Acc: 76.05%
Epoch [6/20], Step [300/391], Loss: 0.6868, Acc: 75.97%
Epoch [6/20] Test Accuracy: 76.37%
Epoch [7/20], Step [100/391], Loss: 0.6331, Acc: 78.13%
Epoch [7/20], Step [200/391], Loss: 0.6028, Acc: 78.39%
Epoch [7/20], Step [300/391], Loss: 0.6409, Acc: 78.22%
Epoch [7/20] Test Accuracy: 76.03%
Epoch [8/20], Step [100/391], Loss: 0.5782, Acc: 79.59%
Epoch [8/20], Step [200/391], Loss: 0.5631, Acc: 80.05%
Epoch [8/20], Step [300/391], Loss: 0.5826, Acc: 79.91%
Epoch [8/20] Test Accuracy: 77.14%
Epoch [9/20], Step [100/391], Loss: 0.5315, Acc: 81.62%
Epoch [9/20], Step [200/391], Loss: 0.5344, Acc: 81.62%
Epoch [9/20], Step [300/391], Loss: 0.5210, Acc: 81.70%
Epoch [9/20] Test Accuracy: 80.37%
Epoch [10/20], Step [100/391], Loss: 0.5117, Acc: 82.30%
Epoch [10/20], Step [200/391], Loss: 0.4795, Acc: 82.93%
Epoch [10/20], Step [300/391], Loss: 0.4883, Acc: 83.04%
Epoch [10/20] Test Accuracy: 78.32%
Epoch [11/20], Step [100/391], Loss: 0.4584, Acc: 84.23%
Epoch [11/20], Step [200/391], Loss: 0.4705, Acc: 83.96%
Epoch [11/20], Step [300/391], Loss: 0.4565, Acc: 84.12%
Epoch [11/20] Test Accuracy: 82.52%
Epoch [12/20], Step [100/391], Loss: 0.4308, Acc: 85.34%
Epoch [12/20], Step [200/391], Loss: 0.4282, Acc: 85.31%
Epoch [12/20], Step [300/391], Loss: 0.4289, Acc: 85.33%
Epoch [12/20] Test Accuracy: 82.54%
Epoch [13/20], Step [100/391], Loss: 0.4030, Acc: 86.14%
Epoch [13/20], Step [200/391], Loss: 0.4017, Acc: 86.18%
Epoch [13/20], Step [300/391], Loss: 0.3956, Acc: 86.26%
Epoch [13/20] Test Accuracy: 82.48%
```

```
Epoch [14/20], Step [100/391], Loss: 0.3707, Acc: 87.27%
Epoch [14/20], Step [200/391], Loss: 0.3840, Acc: 87.12%
Epoch [14/20], Step [300/391], Loss: 0.3788, Acc: 87.07%
Epoch [14/20] Test Accuracy: 84.23%
Epoch [15/20], Step [100/391], Loss: 0.3592, Acc: 87.70%
Epoch [15/20], Step [200/391], Loss: 0.3545, Acc: 87.75%
Epoch [15/20], Step [300/391], Loss: 0.3636, Acc: 87.54%
Epoch [15/20] Test Accuracy: 85.53%
Epoch [16/20], Step [100/391], Loss: 0.3353, Acc: 88.77%
Epoch [16/20], Step [200/391], Loss: 0.3381, Acc: 88.45%
Epoch [16/20], Step [300/391], Loss: 0.3443, Acc: 88.40%
Epoch [16/20] Test Accuracy: 85.65%
Epoch [17/20], Step [100/391], Loss: 0.3234, Acc: 89.17%
Epoch [17/20], Step [200/391], Loss: 0.3313, Acc: 89.00%
Epoch [17/20], Step [300/391], Loss: 0.3172, Acc: 89.05%
Epoch [17/20] Test Accuracy: 86.38%
Epoch [18/20], Step [100/391], Loss: 0.3142, Acc: 89.58%
Epoch [18/20], Step [200/391], Loss: 0.3076, Acc: 89.57%
Epoch [18/20], Step [300/391], Loss: 0.3035, Acc: 89.62%
Epoch [18/20] Test Accuracy: 86.35%
Epoch [19/20], Step [100/391], Loss: 0.3028, Acc: 89.77%
Epoch [19/20], Step [200/391], Loss: 0.3024, Acc: 89.74%
Epoch [19/20], Step [300/391], Loss: 0.3093, Acc: 89.55%
Epoch [19/20] Test Accuracy: 86.53%
Epoch [20/20], Step [100/391], Loss: 0.3000, Acc: 89.91%
Epoch [20/20], Step [200/391], Loss: 0.2920, Acc: 90.08%
Epoch [20/20], Step [300/391], Loss: 0.2957, Acc: 90.01%
Epoch [20/20] Test Accuracy: 86.58%
Best Test Accuracy: 86.58%

Model Performance Metrics:
F1 Score (macro): 0.8652
Recall (macro): 0.8658
Precision (macro): 0.8651
```




In [4]:
```
# 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_sco
import seaborn as sns
import numpy as np

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=str
        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,
        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,
                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=Fals
        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='rel
                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,
```

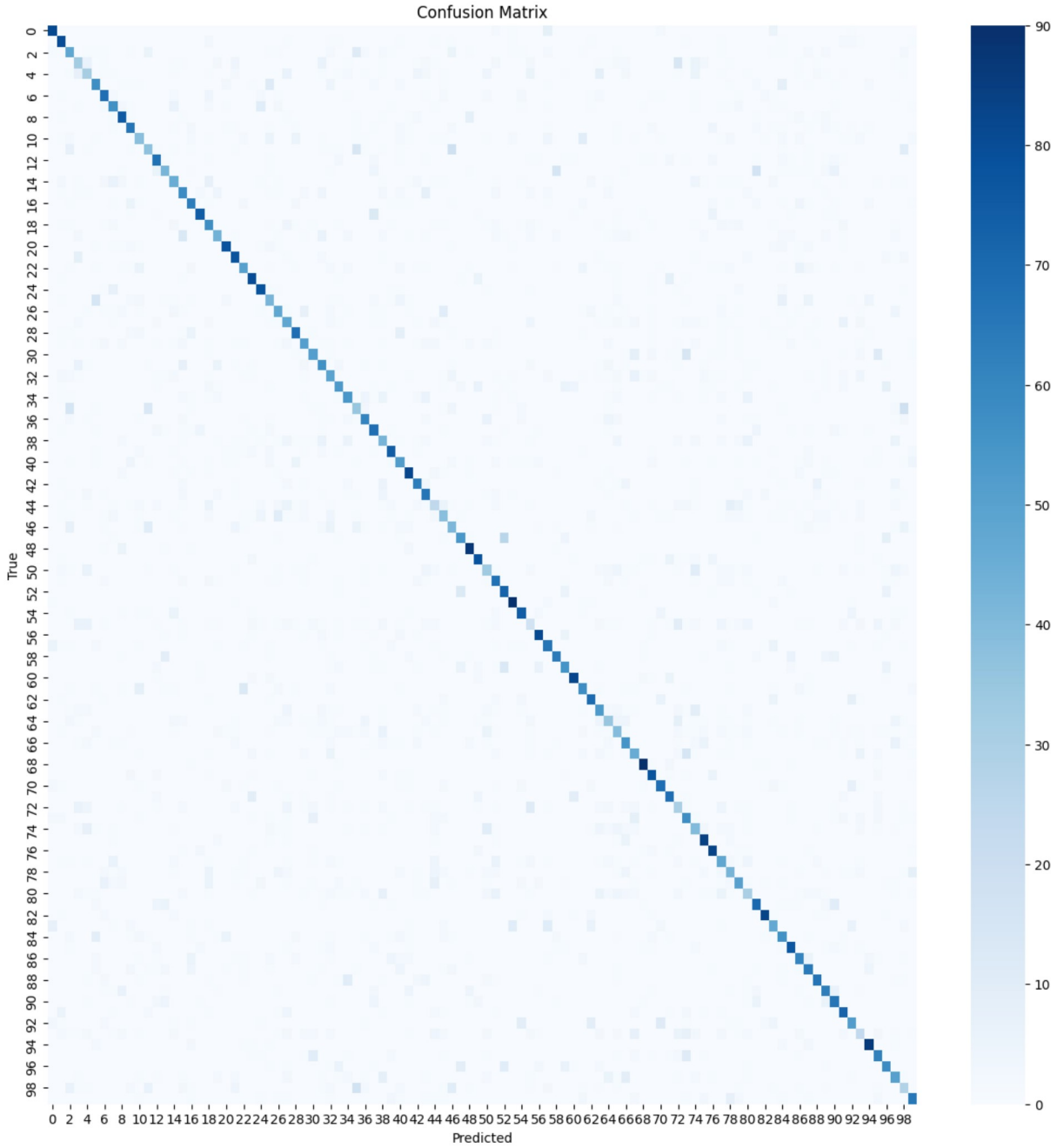
```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True,
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu

# 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
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)
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
    train_losses.append(epoch_loss / len(trainloader))
    train_accs.append(100 * correct / total)
    scheduler.step()

    # Evaluate on test set after each epoch
```

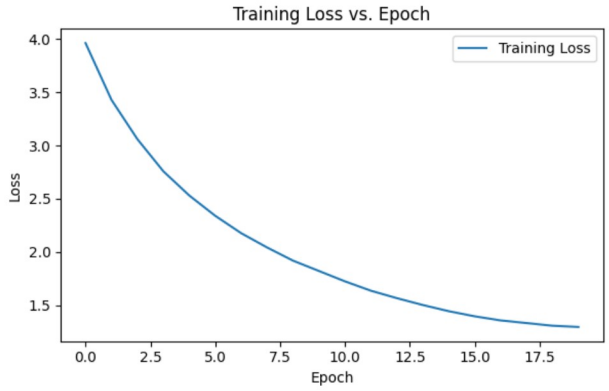
```
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 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}')

# Plot confusion matrix
cmatrix = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(15, 15))
sns.heatmap(cmatrix, 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()
```

```
Files already downloaded and verified
Files already downloaded and verified
Total Parameters: 4,949,412
Trainable Parameters: 4,949,412
Epoch [1/20], Step [100/391], Loss: 4.3557, Acc: 4.27%
Epoch [1/20], Step [200/391], Loss: 3.9958, Acc: 6.55%
Epoch [1/20], Step [300/391], Loss: 3.8001, Acc: 8.03%
Epoch [1/20] Test Accuracy: 12.87%
Epoch [2/20], Step [100/391], Loss: 3.5660, Acc: 15.13%
Epoch [2/20], Step [200/391], Loss: 3.4644, Acc: 15.85%
Epoch [2/20], Step [300/391], Loss: 3.3668, Acc: 16.77%
Epoch [2/20] Test Accuracy: 20.27%
Epoch [3/20], Step [100/391], Loss: 3.1885, Acc: 21.69%
Epoch [3/20], Step [200/391], Loss: 3.1049, Acc: 22.34%
Epoch [3/20], Step [300/391], Loss: 2.9979, Acc: 23.16%
Epoch [3/20] Test Accuracy: 24.97%
Epoch [4/20], Step [100/391], Loss: 2.8422, Acc: 28.44%
Epoch [4/20], Step [200/391], Loss: 2.7708, Acc: 29.14%
Epoch [4/20], Step [300/391], Loss: 2.7434, Acc: 29.36%
Epoch [4/20] Test Accuracy: 29.49%
Epoch [5/20], Step [100/391], Loss: 2.5992, Acc: 32.88%
Epoch [5/20], Step [200/391], Loss: 2.5475, Acc: 33.35%
Epoch [5/20], Step [300/391], Loss: 2.5052, Acc: 33.68%
Epoch [5/20] Test Accuracy: 35.48%
Epoch [6/20], Step [100/391], Loss: 2.4094, Acc: 36.47%
Epoch [6/20], Step [200/391], Loss: 2.3707, Acc: 37.03%
Epoch [6/20], Step [300/391], Loss: 2.3221, Acc: 37.61%
Epoch [6/20] Test Accuracy: 37.82%
Epoch [7/20], Step [100/391], Loss: 2.2125, Acc: 41.44%
Epoch [7/20], Step [200/391], Loss: 2.2005, Acc: 41.46%
Epoch [7/20], Step [300/391], Loss: 2.1597, Acc: 41.71%
Epoch [7/20] Test Accuracy: 42.44%
Epoch [8/20], Step [100/391], Loss: 2.0612, Acc: 45.02%
Epoch [8/20], Step [200/391], Loss: 2.0534, Acc: 44.73%
Epoch [8/20], Step [300/391], Loss: 2.0485, Acc: 44.82%
Epoch [8/20] Test Accuracy: 43.66%
Epoch [9/20], Step [100/391], Loss: 1.9308, Acc: 47.17%
Epoch [9/20], Step [200/391], Loss: 1.9302, Acc: 47.38%
Epoch [9/20], Step [300/391], Loss: 1.9156, Acc: 47.52%
Epoch [9/20] Test Accuracy: 45.42%
Epoch [10/20], Step [100/391], Loss: 1.8179, Acc: 50.05%
Epoch [10/20], Step [200/391], Loss: 1.8148, Acc: 50.08%
Epoch [10/20], Step [300/391], Loss: 1.8415, Acc: 49.96%
Epoch [10/20] Test Accuracy: 46.95%
Epoch [11/20], Step [100/391], Loss: 1.7377, Acc: 52.15%
Epoch [11/20], Step [200/391], Loss: 1.7103, Acc: 52.62%
Epoch [11/20], Step [300/391], Loss: 1.7236, Acc: 52.58%
Epoch [11/20] Test Accuracy: 50.69%
Epoch [12/20], Step [100/391], Loss: 1.6463, Acc: 54.79%
Epoch [12/20], Step [200/391], Loss: 1.6347, Acc: 54.48%
Epoch [12/20], Step [300/391], Loss: 1.6374, Acc: 54.66%
Epoch [12/20] Test Accuracy: 51.05%
Epoch [13/20], Step [100/391], Loss: 1.5672, Acc: 56.73%
Epoch [13/20], Step [200/391], Loss: 1.5701, Acc: 56.67%
Epoch [13/20], Step [300/391], Loss: 1.5818, Acc: 56.35%
Epoch [13/20] Test Accuracy: 52.57%
```

```
Epoch [14/20], Step [100/391], Loss: 1.5204, Acc: 57.82%
Epoch [14/20], Step [200/391], Loss: 1.4894, Acc: 58.19%
Epoch [14/20], Step [300/391], Loss: 1.5046, Acc: 58.13%
Epoch [14/20] Test Accuracy: 53.88%
Epoch [15/20], Step [100/391], Loss: 1.4405, Acc: 59.73%
Epoch [15/20], Step [200/391], Loss: 1.4453, Acc: 59.34%
Epoch [15/20], Step [300/391], Loss: 1.4505, Acc: 59.46%
Epoch [15/20] Test Accuracy: 54.90%
Epoch [16/20], Step [100/391], Loss: 1.3893, Acc: 61.30%
Epoch [16/20], Step [200/391], Loss: 1.3891, Acc: 61.03%
Epoch [16/20], Step [300/391], Loss: 1.4164, Acc: 60.88%
Epoch [16/20] Test Accuracy: 56.05%
Epoch [17/20], Step [100/391], Loss: 1.3433, Acc: 62.50%
Epoch [17/20], Step [200/391], Loss: 1.3661, Acc: 62.11%
Epoch [17/20], Step [300/391], Loss: 1.3423, Acc: 62.23%
Epoch [17/20] Test Accuracy: 57.48%
Epoch [18/20], Step [100/391], Loss: 1.3312, Acc: 62.66%
Epoch [18/20], Step [200/391], Loss: 1.3254, Acc: 62.76%
Epoch [18/20], Step [300/391], Loss: 1.3420, Acc: 62.62%
Epoch [18/20] Test Accuracy: 58.17%
Epoch [19/20], Step [100/391], Loss: 1.3121, Acc: 63.55%
Epoch [19/20], Step [200/391], Loss: 1.3042, Acc: 63.31%
Epoch [19/20], Step [300/391], Loss: 1.3114, Acc: 63.16%
Epoch [19/20] Test Accuracy: 58.06%
Epoch [20/20], Step [100/391], Loss: 1.2933, Acc: 63.34%
Epoch [20/20], Step [200/391], Loss: 1.3147, Acc: 63.47%
Epoch [20/20], Step [300/391], Loss: 1.3017, Acc: 63.60%
Epoch [20/20] Test Accuracy: 58.35%
Best Test Accuracy: 58.35%

Model Performance Metrics:
F1 Score (macro): 0.5789
Recall (macro): 0.5835
Precision (macro): 0.5785
```





In [5]:
```
# 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_sco
import seaborn as sns
import numpy as np

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=str
        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,
        self.bn2 = nn.BatchNorm2d(out_channels)

        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,
                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=Fals
        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)
        )
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)
        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='rel
                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))
])
```

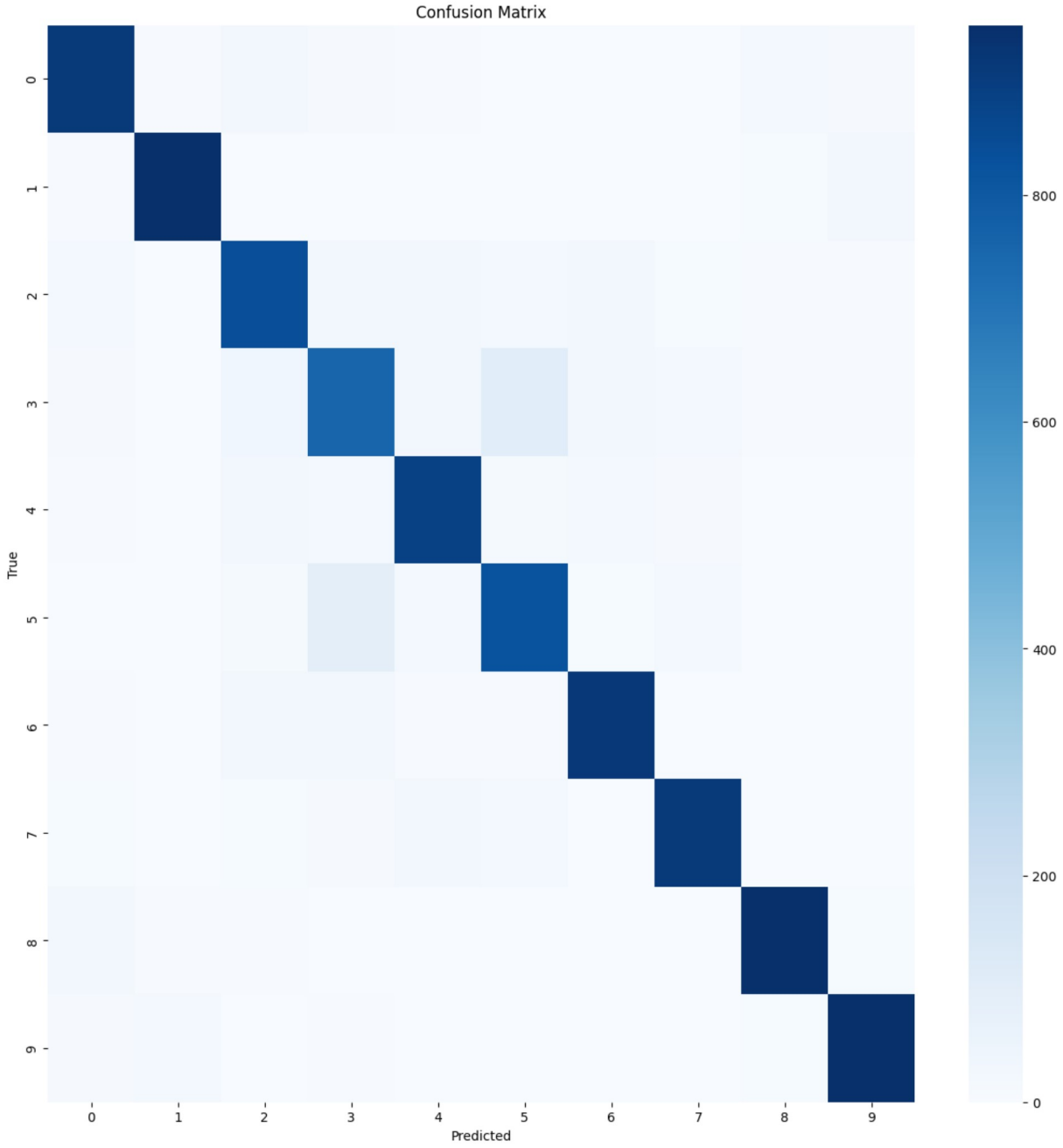
```
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
])

# CIFAR-10 Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, t
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, t
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu
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)
```

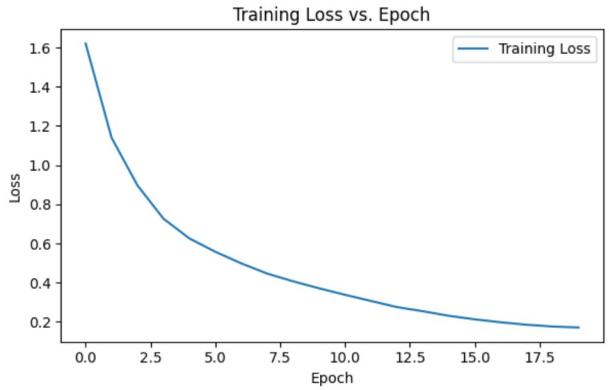
```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        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')
```

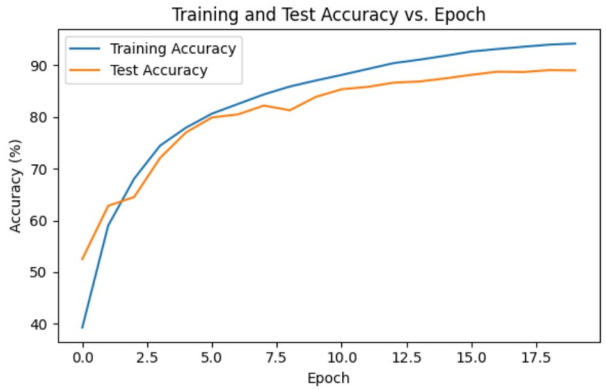
```
print(f'\nModel Performance Metrics:')
print(f'F1 Score (macro): {f1:.4f}')
print(f'Recall (macro): {recall:.4f}')
print(f'Precision (macro): {precision:.4f}')

# 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()
```

```
Files already downloaded and verified
Files already downloaded and verified
Total Parameters: 11,173,962
Trainable Parameters: 11,173,962
Epoch [1/20], Step [100/391], Loss: 1.9668, Acc: 25.55%
Epoch [1/20], Step [200/391], Loss: 1.6524, Acc: 31.31%
Epoch [1/20], Step [300/391], Loss: 1.4887, Acc: 35.94%
Epoch [1/20] Test Accuracy: 52.52%
Epoch [2/20], Step [100/391], Loss: 1.2326, Acc: 55.59%
Epoch [2/20], Step [200/391], Loss: 1.1835, Acc: 56.34%
Epoch [2/20], Step [300/391], Loss: 1.1015, Acc: 57.88%
Epoch [2/20] Test Accuracy: 62.84%
Epoch [3/20], Step [100/391], Loss: 0.9572, Acc: 65.84%
Epoch [3/20], Step [200/391], Loss: 0.9253, Acc: 66.39%
Epoch [3/20], Step [300/391], Loss: 0.8600, Acc: 67.48%
Epoch [3/20] Test Accuracy: 64.54%
Epoch [4/20], Step [100/391], Loss: 0.7671, Acc: 73.02%
Epoch [4/20], Step [200/391], Loss: 0.7526, Acc: 73.24%
Epoch [4/20], Step [300/391], Loss: 0.7020, Acc: 73.95%
Epoch [4/20] Test Accuracy: 72.11%
Epoch [5/20], Step [100/391], Loss: 0.6468, Acc: 77.07%
Epoch [5/20], Step [200/391], Loss: 0.6339, Acc: 77.29%
Epoch [5/20], Step [300/391], Loss: 0.6202, Acc: 77.73%
Epoch [5/20] Test Accuracy: 77.04%
Epoch [6/20], Step [100/391], Loss: 0.5708, Acc: 80.41%
Epoch [6/20], Step [200/391], Loss: 0.5661, Acc: 80.28%
Epoch [6/20], Step [300/391], Loss: 0.5388, Acc: 80.63%
Epoch [6/20] Test Accuracy: 79.93%
Epoch [7/20], Step [100/391], Loss: 0.5076, Acc: 81.80%
Epoch [7/20], Step [200/391], Loss: 0.4969, Acc: 82.24%
Epoch [7/20], Step [300/391], Loss: 0.5018, Acc: 82.38%
Epoch [7/20] Test Accuracy: 80.52%
Epoch [8/20], Step [100/391], Loss: 0.4484, Acc: 84.37%
Epoch [8/20], Step [200/391], Loss: 0.4613, Acc: 84.05%
Epoch [8/20], Step [300/391], Loss: 0.4375, Acc: 84.28%
Epoch [8/20] Test Accuracy: 82.23%
Epoch [9/20], Step [100/391], Loss: 0.4003, Acc: 85.82%
Epoch [9/20], Step [200/391], Loss: 0.4070, Acc: 85.88%
Epoch [9/20], Step [300/391], Loss: 0.4190, Acc: 85.86%
Epoch [9/20] Test Accuracy: 81.32%
Epoch [10/20], Step [100/391], Loss: 0.3556, Acc: 87.85%
Epoch [10/20], Step [200/391], Loss: 0.3775, Acc: 87.34%
Epoch [10/20], Step [300/391], Loss: 0.3791, Acc: 87.11%
Epoch [10/20] Test Accuracy: 83.90%
Epoch [11/20], Step [100/391], Loss: 0.3277, Acc: 88.61%
Epoch [11/20], Step [200/391], Loss: 0.3416, Acc: 88.35%
Epoch [11/20], Step [300/391], Loss: 0.3344, Acc: 88.31%
Epoch [11/20] Test Accuracy: 85.39%
Epoch [12/20], Step [100/391], Loss: 0.3135, Acc: 89.26%
Epoch [12/20], Step [200/391], Loss: 0.3113, Acc: 89.12%
Epoch [12/20], Step [300/391], Loss: 0.3043, Acc: 89.17%
Epoch [12/20] Test Accuracy: 85.85%
Epoch [13/20], Step [100/391], Loss: 0.2724, Acc: 90.64%
Epoch [13/20], Step [200/391], Loss: 0.2838, Acc: 90.42%
Epoch [13/20], Step [300/391], Loss: 0.2708, Acc: 90.46%
Epoch [13/20] Test Accuracy: 86.66%
```

```
Epoch [14/20], Step [100/391], Loss: 0.2635, Acc: 90.69%
Epoch [14/20], Step [200/391], Loss: 0.2457, Acc: 91.03%
Epoch [14/20], Step [300/391], Loss: 0.2491, Acc: 91.18%
Epoch [14/20] Test Accuracy: 86.89%
Epoch [15/20], Step [100/391], Loss: 0.2296, Acc: 91.98%
Epoch [15/20], Step [200/391], Loss: 0.2315, Acc: 92.02%
Epoch [15/20], Step [300/391], Loss: 0.2287, Acc: 91.95%
Epoch [15/20] Test Accuracy: 87.49%
Epoch [16/20], Step [100/391], Loss: 0.2064, Acc: 93.06%
Epoch [16/20], Step [200/391], Loss: 0.2165, Acc: 92.79%
Epoch [16/20], Step [300/391], Loss: 0.2081, Acc: 92.83%
Epoch [16/20] Test Accuracy: 88.18%
Epoch [17/20], Step [100/391], Loss: 0.1898, Acc: 93.48%
Epoch [17/20], Step [200/391], Loss: 0.2023, Acc: 93.23%
Epoch [17/20], Step [300/391], Loss: 0.2027, Acc: 93.15%
Epoch [17/20] Test Accuracy: 88.78%
Epoch [18/20], Step [100/391], Loss: 0.1864, Acc: 93.51%
Epoch [18/20], Step [200/391], Loss: 0.1808, Acc: 93.67%
Epoch [18/20], Step [300/391], Loss: 0.1924, Acc: 93.55%
Epoch [18/20] Test Accuracy: 88.72%
Epoch [19/20], Step [100/391], Loss: 0.1708, Acc: 94.34%
Epoch [19/20], Step [200/391], Loss: 0.1819, Acc: 94.09%
Epoch [19/20], Step [300/391], Loss: 0.1792, Acc: 94.02%
Epoch [19/20] Test Accuracy: 89.09%
Epoch [20/20], Step [100/391], Loss: 0.1666, Acc: 94.50%
Epoch [20/20], Step [200/391], Loss: 0.1749, Acc: 94.27%
Epoch [20/20], Step [300/391], Loss: 0.1692, Acc: 94.25%
Epoch [20/20] Test Accuracy: 89.04%
Best Test Accuracy: 89.09%

Model Performance Metrics:
F1 Score (macro): 0.8902
Recall (macro): 0.8904
Precision (macro): 0.8902
```




In [6]:
```
# 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_sco
import seaborn as sns
import numpy as np
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=str
        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,
        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,
                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=Fals
        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='rel
                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),
```

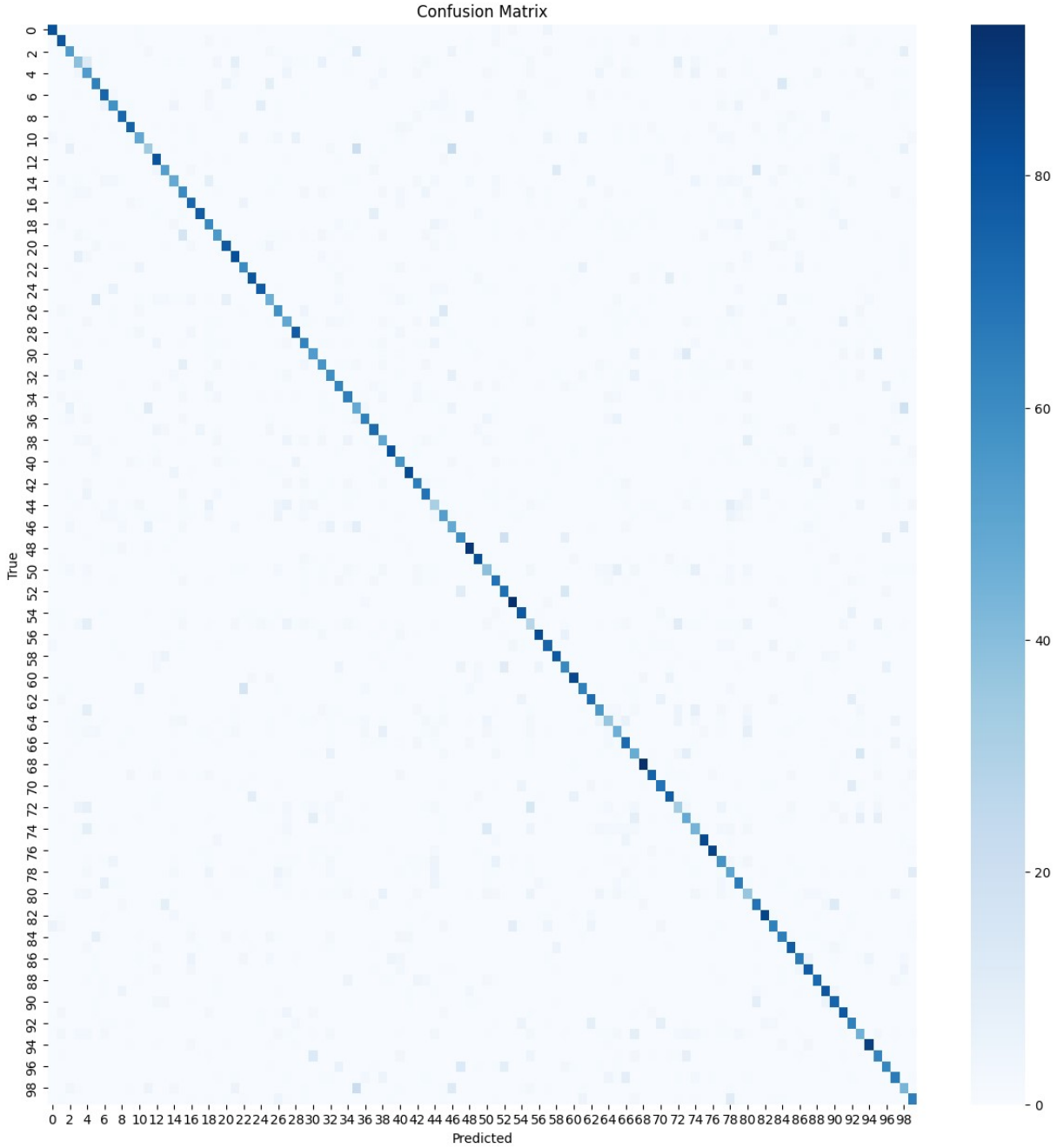
```
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
])

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,
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, n

testset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True,
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, nu

# 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)
```

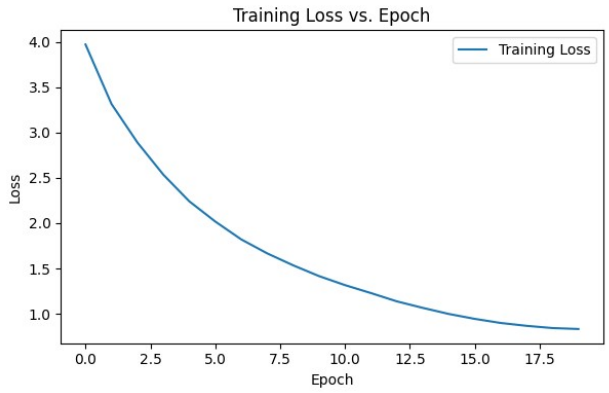
```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        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)
```

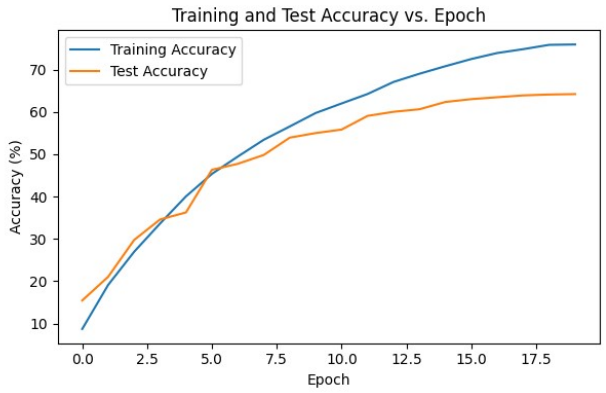
```
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# 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}')

# 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()
```

```
Files already downloaded and verified
Files already downloaded and verified
Total Parameters: 11,220,132
Trainable Parameters: 11,220,132
Epoch [1/20], Step [100/391], Loss: 4.4315, Acc: 3.10%
Epoch [1/20], Step [200/391], Loss: 3.9913, Acc: 5.57%
Epoch [1/20], Step [300/391], Loss: 3.7810, Acc: 7.48%
Epoch [1/20] Test Accuracy: 15.45%
Epoch [2/20], Step [100/391], Loss: 3.4963, Acc: 15.81%
Epoch [2/20], Step [200/391], Loss: 3.3752, Acc: 16.93%
Epoch [2/20], Step [300/391], Loss: 3.2489, Acc: 18.16%
Epoch [2/20] Test Accuracy: 21.04%
Epoch [3/20], Step [100/391], Loss: 3.0389, Acc: 24.28%
Epoch [3/20], Step [200/391], Loss: 2.8948, Acc: 25.34%
Epoch [3/20], Step [300/391], Loss: 2.8424, Acc: 26.22%
Epoch [3/20] Test Accuracy: 29.74%
Epoch [4/20], Step [100/391], Loss: 2.6331, Acc: 31.64%
Epoch [4/20], Step [200/391], Loss: 2.5488, Acc: 32.38%
Epoch [4/20], Step [300/391], Loss: 2.5108, Acc: 32.94%
Epoch [4/20] Test Accuracy: 34.59%
Epoch [5/20], Step [100/391], Loss: 2.3146, Acc: 38.40%
Epoch [5/20], Step [200/391], Loss: 2.2808, Acc: 38.95%
Epoch [5/20], Step [300/391], Loss: 2.2164, Acc: 39.41%
Epoch [5/20] Test Accuracy: 36.25%
Epoch [6/20], Step [100/391], Loss: 2.0757, Acc: 43.91%
Epoch [6/20], Step [200/391], Loss: 2.0415, Acc: 44.29%
Epoch [6/20], Step [300/391], Loss: 1.9931, Acc: 44.88%
Epoch [6/20] Test Accuracy: 46.33%
Epoch [7/20], Step [100/391], Loss: 1.8434, Acc: 48.82%
Epoch [7/20], Step [200/391], Loss: 1.8293, Acc: 49.27%
Epoch [7/20], Step [300/391], Loss: 1.8062, Acc: 49.36%
Epoch [7/20] Test Accuracy: 47.75%
Epoch [8/20], Step [100/391], Loss: 1.6749, Acc: 53.20%
Epoch [8/20], Step [200/391], Loss: 1.6652, Acc: 53.33%
Epoch [8/20], Step [300/391], Loss: 1.6735, Acc: 53.35%
Epoch [8/20] Test Accuracy: 49.84%
Epoch [9/20], Step [100/391], Loss: 1.5202, Acc: 57.13%
Epoch [9/20], Step [200/391], Loss: 1.5215, Acc: 56.94%
Epoch [9/20], Step [300/391], Loss: 1.5618, Acc: 56.63%
Epoch [9/20] Test Accuracy: 53.93%
Epoch [10/20], Step [100/391], Loss: 1.4058, Acc: 60.20%
Epoch [10/20], Step [200/391], Loss: 1.4391, Acc: 59.64%
Epoch [10/20], Step [300/391], Loss: 1.3953, Acc: 59.82%
Epoch [10/20] Test Accuracy: 55.01%
Epoch [11/20], Step [100/391], Loss: 1.3058, Acc: 62.52%
Epoch [11/20], Step [200/391], Loss: 1.3248, Acc: 62.03%
Epoch [11/20], Step [300/391], Loss: 1.3130, Acc: 62.16%
Epoch [11/20] Test Accuracy: 55.84%
Epoch [12/20], Step [100/391], Loss: 1.2292, Acc: 64.10%
Epoch [12/20], Step [200/391], Loss: 1.2200, Acc: 64.13%
Epoch [12/20], Step [300/391], Loss: 1.2335, Acc: 64.19%
Epoch [12/20] Test Accuracy: 59.07%
Epoch [13/20], Step [100/391], Loss: 1.1336, Acc: 67.05%
Epoch [13/20], Step [200/391], Loss: 1.1406, Acc: 67.14%
Epoch [13/20], Step [300/391], Loss: 1.1274, Acc: 67.22%
Epoch [13/20] Test Accuracy: 60.04%
```

```
Epoch [14/20], Step [100/391], Loss: 1.0553, Acc: 69.25%
Epoch [14/20], Step [200/391], Loss: 1.0718, Acc: 69.01%
Epoch [14/20], Step [300/391], Loss: 1.0637, Acc: 69.15%
Epoch [14/20] Test Accuracy: 60.64%
Epoch [15/20], Step [100/391], Loss: 1.0037, Acc: 70.73%
Epoch [15/20], Step [200/391], Loss: 0.9937, Acc: 71.08%
Epoch [15/20], Step [300/391], Loss: 1.0135, Acc: 70.61%
Epoch [15/20] Test Accuracy: 62.36%
Epoch [16/20], Step [100/391], Loss: 0.9534, Acc: 72.41%
Epoch [16/20], Step [200/391], Loss: 0.9334, Acc: 72.53%
Epoch [16/20], Step [300/391], Loss: 0.9555, Acc: 72.38%
Epoch [16/20] Test Accuracy: 63.03%
Epoch [17/20], Step [100/391], Loss: 0.9027, Acc: 74.06%
Epoch [17/20], Step [200/391], Loss: 0.8828, Acc: 74.36%
Epoch [17/20], Step [300/391], Loss: 0.9082, Acc: 74.04%
Epoch [17/20] Test Accuracy: 63.48%
Epoch [18/20], Step [100/391], Loss: 0.8574, Acc: 74.94%
Epoch [18/20], Step [200/391], Loss: 0.8781, Acc: 74.62%
Epoch [18/20], Step [300/391], Loss: 0.8672, Acc: 74.78%
Epoch [18/20] Test Accuracy: 63.91%
Epoch [19/20], Step [100/391], Loss: 0.8546, Acc: 75.98%
Epoch [19/20], Step [200/391], Loss: 0.8355, Acc: 76.05%
Epoch [19/20], Step [300/391], Loss: 0.8330, Acc: 75.96%
Epoch [19/20] Test Accuracy: 64.12%
Epoch [20/20], Step [100/391], Loss: 0.8354, Acc: 75.80%
Epoch [20/20], Step [200/391], Loss: 0.8312, Acc: 75.96%
Epoch [20/20], Step [300/391], Loss: 0.8260, Acc: 75.98%
Epoch [20/20] Test Accuracy: 64.22%
Best Test Accuracy: 64.22%

Model Performance Metrics:
F1 Score (macro): 0.6407
Recall (macro): 0.6422
Precision (macro): 0.6430
```


Confusion Matrix
True
Predicted
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98
0
20
40
60
80


Training Loss vs. Epoch
Training Loss
Loss
Epoch
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5


Training and Test Accuracy vs. Epoch
Training Accuracy
Test Accuracy
Accuracy (%)
Epoch
70
60
50
40
30
20
10
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5