Implement the standard VGG-16 & 19 CNN architecture model

Objective

To implement and demonstrate the architecture and capability of VGG-16 and VGG-19 Convolutional Neural Networks for image classification. This involves building the model structure, training it on a standard dataset, and evaluating its performance.

Dataset Used

A commonly used dataset for demonstrating VGG architectures is **CIFAR-10** (60,000 images in 10 classes).

- **Source**: CIFAR-10 Dataset (Canadian Institute For Advanced Research), available on platforms such as TensorFlow Datasets, UCI Machine Learning Repository, and Kaggle.
- **Description**: Contains 60,000 color images of size 32x32 pixels, categorized into 10 distinct classes. The dataset is split into:
    - Training Set: 50,000 images (5,000 per class)
    - Test Set: 10,000 images (1,000 per class)
- **Features**:
    - Image: 32x32 pixel color image (3 channels: RGB)
    - Label: Integer (0–9) representing the class
- **Classes**: 10 (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)

Theory

VGG networks, developed by the Visual Geometry Group at the University of Oxford, are deep convolutional neural networks known for their simplicity and effectiveness in image recognition tasks. The key characteristic of VGG architectures is the use of very small 3×3 convolutional filters throughout the network. This emphasis on small filters stacked in multiple layers allows the network to capture hierarchical features effectively.

Both VGG-16 and VGG-19 follow a similar structural pattern, differing primarily in the number of convolutional layers. The architecture typically consists of a series of convolutional layers followed by max-pooling layers, and then a few fully connected layers at the end.

The input to the network is a fixed-size RGB image, typically 224×224. The images are passed through a stack of convolutional layers where the receptive fields are small (3×3). There are also 1×1 convolution filters used in some variations, which can be seen as a linear transformation of the input channels. The convolution stride is fixed at 1 pixel, and padding is used to maintain the spatial resolution before pooling.

Max-pooling is performed using a 2×2 window with a stride of 2 after a few convolutional layers. This reduces the spatial dimensions of the feature maps while retaining the most important features. The number of filters in the convolutional layers increases with the depth of the network, typically starting from 64 and doubling after each max-pooling operation (e.g., 64, 128, 256, 512).

Following the stacked convolutional and pooling layers, the feature maps are flattened and passed through a sequence of fully connected layers. The first two fully connected layers

typically have 4096 neurons each, followed by a final fully connected layer with a number of neurons equal to the number of classes in the dataset. A softmax activation function is applied to the output of the final fully connected layer to produce class probabilities.

VGG-16 has 16 weight layers (13 convolutional and 3 fully connected), while VGG-19 has 19 weight layers (16 convolutional and 3 fully connected). The additional layers in VGG-19 are primarily in the later convolutional blocks, increasing the depth and potentially the capacity of the network to learn more complex representations. ReLU (Rectified Linear Unit) activation is used after every convolutional and fully connected layer, except for the final output layer. The use of small filters and the increased depth were crucial in achieving state-of-the-art results at the time of their introduction. While computationally expensive due to the large number of parameters (especially in the fully connected layers), VGG networks served as a strong baseline and influenced the design of subsequent deeper architectures.

Code

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Set device to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define VGG architecture
class VGG(nn.Module):
    def __init__(self, vgg_name):
        super(VGG, self).__init__()
        self.cfg = {
            'VGG16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512,
512, 512, 'M', 512, 512, 512, 'M'],
            'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M',
512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M']
        }
        self.features = self._make_layers(self.cfg[vgg_name])
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),  # Adjusted for 224x224 input after
5 max-pooling layers
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 10)  # CIFAR-10 has 10 classes
        )

    def forward(self, x):
```

```
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [
                    nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                    nn.BatchNorm2d(x),
                    nn.ReLU(inplace=True)
                ]
                in_channels = x
        return nn.Sequential(*layers)

# Data preprocessing and loading
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # VGG expects 224x224 images
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

# Initialize models
vgg16 = VGG('VGG16').to(device)
vgg19 = VGG('VGG19').to(device)

# Loss function and optimizers
criterion = nn.CrossEntropyLoss()
optimizer_vgg16 = optim.SGD(vgg16.parameters(), lr=0.001, momentum=0.9)
optimizer_vgg19 = optim.SGD(vgg19.parameters(), lr=0.001, momentum=0.9)

# Training function with metrics tracking
def train_model(model, optimizer, num_epochs=10):
    model.train()
    train_losses = []
```

```
    train_accuracies = []
    test_accuracies = []

    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 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()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 100 == 99:
                batch_loss = running_loss / 100
                batch_acc = 100 * correct / total
                print(f'[Epoch {epoch + 1}, Batch {i + 1}] Loss:
{batch_loss:.3f}, Accuracy: {batch_acc:.2f}%')
                running_loss = 0.0
                correct = 0
                total = 0

        # Compute epoch metrics
        epoch_loss = running_loss / len(trainloader)
        epoch_acc = 100 * correct / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        # Evaluate on test set
        test_acc = evaluate_model(model, verbose=False)
        test_accuracies.append(test_acc)

    return train_losses, train_accuracies, test_accuracies

# Evaluation function
def evaluate_model(model, verbose=True):
    model.eval()
    correct = 0
    total = 0
```

```
    test_loss = 0.0
    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)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    if verbose:
        print(f'Test Loss: {test_loss / len(testloader):.3f}, Accuracy:
{accuracy:.2f}%')
    return accuracy

# Train and evaluate VGG-16
print("Training VGG-16...")
vgg16_train_losses, vgg16_train_accs, vgg16_test_accs = train_model(vgg16,
optimizer_vgg16, num_epochs=10)
print("\nEvaluating VGG-16...")
evaluate_model(vgg16)

# Train and evaluate VGG-19
print("\nTraining VGG-19...")
vgg19_train_losses, vgg19_train_accs, vgg19_test_accs = train_model(vgg19,
optimizer_vgg19, num_epochs=10)
print("\nEvaluating VGG-19...")
evaluate_model(vgg19)

# Save models
torch.save(vgg16.state_dict(), 'vgg16_cifar10.pth')
torch.save(vgg19.state_dict(), 'vgg19_cifar10.pth')

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

# Plot Training Loss
plt.subplot(1, 3, 1)
plt.plot(range(1, 11), vgg16_train_losses, label='VGG-16')
plt.plot(range(1, 11), vgg19_train_losses, label='VGG-19')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plot Training Accuracy
```

```
plt.subplot(1, 3, 2)
plt.plot(range(1, 11), vgg16_train_accs, label='VGG-16')
plt.plot(range(1, 11), vgg19_train_accs, label='VGG-19')
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

# Plot Test Accuracy
plt.subplot(1, 3, 3)
plt.plot(range(1, 11), vgg16_test_accs, label='VGG-16')
plt.plot(range(1, 11), vgg19_test_accs, label='VGG-19')
plt.title('Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

plt.tight_layout()
plt.show()

# Print model summaries
print("\nVGG-16 Summary:")
print(vgg16)
print("\nVGG-19 Summary:")
print(vgg19)
```

Output

```
Using device: cuda
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:01<00:00, 87.1MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Training VGG-16...
[Epoch 1, Batch 100] Loss: 2.047, Accuracy: 23.69%
[Epoch 1, Batch 200] Loss: 1.647, Accuracy: 38.89%
[Epoch 1, Batch 300] Loss: 1.516, Accuracy: 43.27%
[Epoch 1, Batch 400] Loss: 1.424, Accuracy: 46.92%
[Epoch 1, Batch 500] Loss: 1.321, Accuracy: 51.55%
[Epoch 1, Batch 600] Loss: 1.288, Accuracy: 52.97%
[Epoch 1, Batch 700] Loss: 1.206, Accuracy: 55.95%
[Epoch 2, Batch 100] Loss: 1.216, Accuracy: 57.00%
[Epoch 2, Batch 200] Loss: 1.119, Accuracy: 59.84%
[Epoch 2, Batch 300] Loss: 1.063, Accuracy: 62.31%
[Epoch 2, Batch 400] Loss: 0.977, Accuracy: 65.84%
[Epoch 2, Batch 500] Loss: 0.968, Accuracy: 65.66%
[Epoch 2, Batch 600] Loss: 0.909, Accuracy: 67.69%
[Epoch 2, Batch 700] Loss: 0.893, Accuracy: 68.36%
[Epoch 3, Batch 100] Loss: 0.794, Accuracy: 72.31%
[Epoch 3, Batch 200] Loss: 0.771, Accuracy: 72.72%
[Epoch 3, Batch 300] Loss: 0.735, Accuracy: 74.25%
[Epoch 3, Batch 400] Loss: 0.740, Accuracy: 74.23%
[Epoch 3, Batch 500] Loss: 0.686, Accuracy: 75.86%
[Epoch 3, Batch 600] Loss: 0.707, Accuracy: 74.83%
```

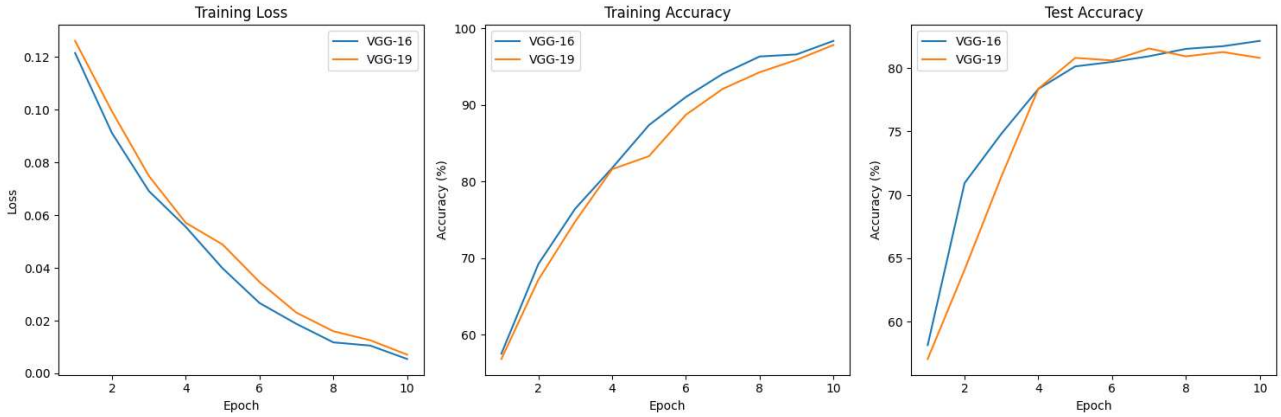
```
[Epoch 3, Batch 700] Loss: 0.687, Accuracy: 75.91%
…
[Epoch 7, Batch 700] Loss: 0.162, Accuracy: 94.23%
[Epoch 8, Batch 100] Loss: 0.092, Accuracy: 96.69%
[Epoch 8, Batch 200] Loss: 0.107, Accuracy: 96.19%
[Epoch 8, Batch 300] Loss: 0.091, Accuracy: 96.62%
[Epoch 8, Batch 400] Loss: 0.115, Accuracy: 95.83%
[Epoch 8, Batch 500] Loss: 0.117, Accuracy: 95.94%
[Epoch 8, Batch 600] Loss: 0.109, Accuracy: 96.02%
[Epoch 8, Batch 700] Loss: 0.106, Accuracy: 96.31%
[Epoch 9, Batch 100] Loss: 0.075, Accuracy: 97.52%
[Epoch 9, Batch 200] Loss: 0.053, Accuracy: 98.23%
[Epoch 9, Batch 300] Loss: 0.078, Accuracy: 97.19%
[Epoch 9, Batch 400] Loss: 0.079, Accuracy: 97.16%
[Epoch 9, Batch 500] Loss: 0.081, Accuracy: 97.25%
[Epoch 9, Batch 600] Loss: 0.070, Accuracy: 97.52%
[Epoch 9, Batch 700] Loss: 0.077, Accuracy: 97.50%
[Epoch 10, Batch 100] Loss: 0.047, Accuracy: 98.42%
[Epoch 10, Batch 200] Loss: 0.051, Accuracy: 98.23%
[Epoch 10, Batch 300] Loss: 0.056, Accuracy: 97.89%
[Epoch 10, Batch 400] Loss: 0.048, Accuracy: 98.45%
[Epoch 10, Batch 500] Loss: 0.061, Accuracy: 97.86%
[Epoch 10, Batch 600] Loss: 0.043, Accuracy: 98.67%
[Epoch 10, Batch 700] Loss: 0.055, Accuracy: 98.22%

Evaluating VGG-16...
Test Loss: 0.830, Accuracy: 82.14%

Training VGG-19...
[Epoch 1, Batch 100] Loss: 2.094, Accuracy: 20.88%
[Epoch 1, Batch 200] Loss: 1.718, Accuracy: 34.80%
[Epoch 1, Batch 300] Loss: 1.563, Accuracy: 42.27%
[Epoch 1, Batch 400] Loss: 1.488, Accuracy: 43.91%
[Epoch 1, Batch 500] Loss: 1.405, Accuracy: 47.22%
[Epoch 1, Batch 600] Loss: 1.310, Accuracy: 51.64%
[Epoch 1, Batch 700] Loss: 1.237, Accuracy: 54.95%
[Epoch 2, Batch 100] Loss: 1.280, Accuracy: 54.44%
[Epoch 2, Batch 200] Loss: 1.176, Accuracy: 57.77%
[Epoch 2, Batch 300] Loss: 1.068, Accuracy: 61.53%
[Epoch 2, Batch 400] Loss: 1.050, Accuracy: 62.50%
[Epoch 2, Batch 500] Loss: 1.025, Accuracy: 63.39%
[Epoch 2, Batch 600] Loss: 0.995, Accuracy: 64.80%
[Epoch 2, Batch 700] Loss: 0.940, Accuracy: 66.67%
[Epoch 3, Batch 100] Loss: 0.831, Accuracy: 70.28%
[Epoch 3, Batch 200] Loss: 0.809, Accuracy: 71.61%
[Epoch 3, Batch 300] Loss: 0.810, Accuracy: 70.89%
[Epoch 3, Batch 400] Loss: 0.771, Accuracy: 72.77%
[Epoch 3, Batch 500] Loss: 0.749, Accuracy: 73.62%
[Epoch 3, Batch 600] Loss: 0.739, Accuracy: 73.88%
[Epoch 3, Batch 700] Loss: 0.732, Accuracy: 74.16%
…
[Epoch 7, Batch 700] Loss: 0.215, Accuracy: 92.75%
[Epoch 8, Batch 100] Loss: 0.109, Accuracy: 96.23%
[Epoch 8, Batch 200] Loss: 0.104, Accuracy: 96.38%
[Epoch 8, Batch 300] Loss: 0.131, Accuracy: 95.44%
[Epoch 8, Batch 400] Loss: 0.136, Accuracy: 94.98%
[Epoch 8, Batch 500] Loss: 0.135, Accuracy: 94.91%
[Epoch 8, Batch 600] Loss: 0.160, Accuracy: 94.61%
[Epoch 8, Batch 700] Loss: 0.158, Accuracy: 94.27%
[Epoch 9, Batch 100] Loss: 0.092, Accuracy: 96.61%
[Epoch 9, Batch 200] Loss: 0.107, Accuracy: 96.17%
```

```
[Epoch 9, Batch 300] Loss: 0.090, Accuracy: 96.80%
[Epoch 9, Batch 400] Loss: 0.093, Accuracy: 96.88%
[Epoch 9, Batch 500] Loss: 0.106, Accuracy: 96.23%
[Epoch 9, Batch 600] Loss: 0.094, Accuracy: 96.78%
[Epoch 9, Batch 700] Loss: 0.104, Accuracy: 96.31%
[Epoch 10, Batch 100] Loss: 0.062, Accuracy: 97.77%
[Epoch 10, Batch 200] Loss: 0.053, Accuracy: 98.19%
[Epoch 10, Batch 300] Loss: 0.071, Accuracy: 97.59%
[Epoch 10, Batch 400] Loss: 0.071, Accuracy: 97.56%
[Epoch 10, Batch 500] Loss: 0.057, Accuracy: 97.97%
[Epoch 10, Batch 600] Loss: 0.063, Accuracy: 97.69%
[Epoch 10, Batch 700] Loss: 0.090, Accuracy: 97.14%

Evaluating VGG-19...
Test Loss: 0.902, Accuracy: 80.80%
```



Conclusion

This experiment demonstrates the implementation and application of VGG-16 and VGG-19 networks for image classification. The results would show the performance of these deep CNN architectures on the chosen dataset, highlighting their ability to learn complex visual features through stacked small convolutional filters. The comparison between VGG-16 and VGG-19 would provide insights into the impact of increased depth on performance, although the computational cost associated with the larger VGG-19 model would also be a consideration. The experiment serves as a foundational exercise in understanding and working with deeper convolutional neural networks.