COMP5541 Assignment 1 - Question 3: Transfer Learning

Tasks:

- Fine-tune ImageNet pre-trained AlexNet with different amounts of training data (10%, 20%, 50%)
- Compare with training from scratch
- Fine-tune other pre-trained models (ResNet18 and VGG16)
- Explore fine-tuning only portions of network layers

In [1]:
```
## Import Required Libraries

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import time
import random
from torch.utils.data import Subset
from torchvision.models import alexnet, AlexNet_Weights
from torchvision.models import vgg16, VGG16_Weights
from torchvision.models import resnet18, ResNet18_Weights
```

In [2]:
```
## Setup Environment

# Set random seed for reproducibility
seed = 42
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)
torch.backends.cudnn.deterministic = True

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

```
Using device: cuda
```

In [3]:
```
## Data Preparation and Loading

# Data Preparation
def load_cifar10(batch_size=128):
    # Define transforms for training data
    # Note: CIFAR-10 images are 32x32 but models like AlexNet expect 224x22
4
    # We'll resize the images to match the expected input size
    transform_train = transforms.Compose([
        transforms.Resize(224),
        transforms.RandomCrop(224, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2
010)),
    ])

    transform_test = transforms.Compose([
        transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2
010)),
    ])

    # Load CIFAR-10 dataset
    trainset = torchvision.datasets.CIFAR10(
        root='./data', train=True, download=True, transform=transform_trai
n)
    testset = torchvision.datasets.CIFAR10(
        root='./data', train=False, download=True, transform=transform_tes
t)

    # Create data loaders
    trainloader = torch.utils.data.DataLoader(
        trainset, batch_size=batch_size, shuffle=True, num_workers=2)
    testloader = torch.utils.data.DataLoader(
        testset, batch_size=batch_size, shuffle=False, num_workers=2)

    # CIFAR-10 classes
    classes = ('plane', 'car', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck')

    return trainset, testset, trainloader, testloader, classes

# Create subset of training data with specific percentage
def create_subset(dataset, percentage):
    """
    Create a subset of the dataset with a given percentage of samples.
    """
    dataset_size = len(dataset)
    subset_size = int(dataset_size * percentage)

    # Generate random indices
    indices = torch.randperm(dataset_size)[:subset_size]

    # Create subset
    subset = Subset(dataset, indices)

    return subset
```

```
# Load the full dataset
trainset, testset, _, testloader, classes = load_cifar10()

print(f"Training set size: {len(trainset)}")
print(f"Test set size: {len(testset)}")
print(f"Classes: {classes}")
```

```
Files already downloaded and verified
Files already downloaded and verified
Training set size: 50000
Test set size: 10000
Classes: ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck')
```

In [4]:

```
## Data Visualization

# Visualize some examples from the dataset
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.figure(figsize=(10, 4))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')
    plt.show()

# Get random training images
dataiter = iter(torch.utils.data.DataLoader(trainset, batch_size=8, shuffle
=True))
images, labels = next(dataiter)

# Show images
imshow(torchvision.utils.make_grid(images))
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(8)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers).
```

```
frog  plane deer  car   bird  horse truck deer
```

In [5]:
```
## Training and Evaluation Functions

# Define training and evaluation functions
def train(model, trainloader, criterion, optimizer, epochs=10):
    model.train()
    train_loss_history = []
    train_acc_history = []

    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0

        start_time = time.time()

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

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Calculate statistics
            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

        # Calculate epoch statistics
        epoch_loss = running_loss / len(trainloader)
        epoch_acc = 100. * correct / total
        epoch_time = time.time() - start_time

        train_loss_history.append(epoch_loss)
        train_acc_history.append(epoch_acc)

        print(f'Epoch {epoch+1}/{epochs} | Loss: {epoch_loss:.4f} | Acc: {e
poch_acc:.2f}% | Time: {epoch_time:.2f}s')

    return train_loss_history, train_acc_history

def evaluate(model, testloader):
    model.eval()
    correct = 0
    total = 0
    test_loss = 0
    criterion = nn.CrossEntropyLoss()

    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()

    test_acc = 100 * correct / total
    test_loss = test_loss / len(testloader)

    print(f'Test Loss: {test_loss:.4f} | Test Acc: {test_acc:.2f}%')
    return test_loss, test_acc
```

Part A: Fine-tuning AlexNet with Different Amounts of Training Data

Fine-tune ImageNet pre-trained AlexNet models on the CIFAR-10 dataset with different amounts of training data (10%, 20%, and 50%) and compare with training from scratch.

In [6]:
```
## Part A Implementation: AlexNet with Different Data Percentages

# Define a function to create AlexNet model
def create_alexnet(pretrained=True):
    if pretrained:
        # Load pre-trained AlexNet
        model = alexnet(weights=AlexNet_Weights.IMAGENET1K_V1)
        print("Loaded pre-trained AlexNet")
    else:
        # Create AlexNet from scratch
        model = alexnet(weights=None)
        print("Created AlexNet from scratch")

    # Modify the classifier for CIFAR-10 (10 classes)
    in_features = model.classifier[6].in_features
    model.classifier[6] = nn.Linear(in_features, 10)

    return model.to(device)

# Define percentages to test
percentages = [0.1, 0.2, 0.5]  # 10%, 20%, 50%
batch_size = 64
epochs = 10

# Results storage
results = {
    'pretrained': {'acc': [], 'loss': []},
    'scratch': {'acc': [], 'loss': []}
}

for percentage in percentages:
    print(f"\n{'='*50}")
    print(f"Training with {percentage*100:.0f}% of data")
    print(f"{'='*50}")

    # Create subset of training data
    subset = create_subset(trainset, percentage)
    trainloader = torch.utils.data.DataLoader(subset, batch_size=batch_siz
e, shuffle=True, num_workers=2)

    print(f"Subset size: {len(subset)} samples")

    # 1. Fine-tune pre-trained AlexNet
    print("\nFine-tuning pre-trained AlexNet:")
    model_pretrained = create_alexnet(pretrained=True)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model_pretrained.parameters(), lr=0.001, momentum
=0.9)

    # Train the model
    train(model_pretrained, trainloader, criterion, optimizer, epochs=epoch
s)

    # Evaluate on test set
    test_loss, test_acc = evaluate(model_pretrained, testloader)
    results['pretrained']['acc'].append(test_acc)
    results['pretrained']['loss'].append(test_loss)

    # 2. Train AlexNet from scratch
    print("\nTraining AlexNet from scratch:")
```

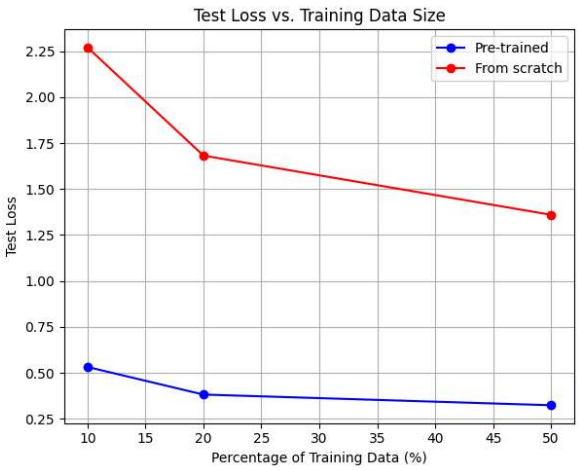
```
    model_scratch = create_alexnet(pretrained=False)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.
9)

    # Train the model
    train(model_scratch, trainloader, criterion, optimizer, epochs=epochs)

    # Evaluate on test set
    test_loss, test_acc = evaluate(model_scratch, testloader)
    results['scratch']['acc'].append(test_acc)
    results['scratch']['loss'].append(test_loss)
```

```
==================================================
Training with 10% of data
==================================================
Subset size: 5000 samples

Fine-tuning pre-trained AlexNet:

Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pt
h" to C:\Users\EverGarden13/.cache\torch\hub\checkpoints\alexnet-owt-7be5b
e79.pth
100%|██████████| 233M/233M [00:02<00:00, 117MB/s]
```

```
Loaded pre-trained AlexNet
Epoch 1/10 | Loss: 1.1414 | Acc: 59.68% | Time: 11.42s
Epoch 2/10 | Loss: 0.6951 | Acc: 75.52% | Time: 7.39s
Epoch 3/10 | Loss: 0.5420 | Acc: 81.50% | Time: 7.37s
Epoch 4/10 | Loss: 0.4835 | Acc: 83.22% | Time: 7.43s
Epoch 5/10 | Loss: 0.3922 | Acc: 86.78% | Time: 7.38s
Epoch 6/10 | Loss: 0.3522 | Acc: 87.60% | Time: 7.40s
Epoch 7/10 | Loss: 0.3254 | Acc: 88.64% | Time: 7.40s
Epoch 8/10 | Loss: 0.3000 | Acc: 89.36% | Time: 7.43s
Epoch 9/10 | Loss: 0.2437 | Acc: 91.38% | Time: 7.56s
Epoch 10/10 | Loss: 0.2157 | Acc: 92.82% | Time: 7.41s
Test Loss: 0.5329 | Test Acc: 82.99%

Training AlexNet from scratch:
Created AlexNet from scratch
Epoch 1/10 | Loss: 2.3024 | Acc: 10.44% | Time: 7.65s
Epoch 2/10 | Loss: 2.3018 | Acc: 12.14% | Time: 7.60s
Epoch 3/10 | Loss: 2.3013 | Acc: 11.26% | Time: 7.38s
Epoch 4/10 | Loss: 2.3004 | Acc: 12.22% | Time: 7.34s
Epoch 5/10 | Loss: 2.2999 | Acc: 12.66% | Time: 7.42s
Epoch 6/10 | Loss: 2.2991 | Acc: 13.08% | Time: 7.38s
Epoch 7/10 | Loss: 2.2972 | Acc: 13.40% | Time: 7.39s
Epoch 8/10 | Loss: 2.2949 | Acc: 11.74% | Time: 7.42s
Epoch 9/10 | Loss: 2.2906 | Acc: 10.24% | Time: 7.41s
Epoch 10/10 | Loss: 2.2823 | Acc: 11.82% | Time: 7.44s
Test Loss: 2.2701 | Test Acc: 12.89%

==================================================
Training with 20% of data
==================================================
Subset size: 10000 samples

Fine-tuning pre-trained AlexNet:
Loaded pre-trained AlexNet
Epoch 1/10 | Loss: 0.9535 | Acc: 66.71% | Time: 10.03s
Epoch 2/10 | Loss: 0.5866 | Acc: 79.74% | Time: 9.89s
Epoch 3/10 | Loss: 0.4826 | Acc: 83.58% | Time: 9.77s
Epoch 4/10 | Loss: 0.4178 | Acc: 85.43% | Time: 9.87s
Epoch 5/10 | Loss: 0.3715 | Acc: 87.13% | Time: 9.89s
Epoch 6/10 | Loss: 0.3321 | Acc: 88.05% | Time: 9.87s
Epoch 7/10 | Loss: 0.2973 | Acc: 89.86% | Time: 9.78s
Epoch 8/10 | Loss: 0.2615 | Acc: 90.64% | Time: 9.97s
Epoch 9/10 | Loss: 0.2576 | Acc: 90.92% | Time: 9.82s
Epoch 10/10 | Loss: 0.2113 | Acc: 92.50% | Time: 9.80s
Test Loss: 0.3830 | Test Acc: 87.18%

Training AlexNet from scratch:
Created AlexNet from scratch
Epoch 1/10 | Loss: 2.3026 | Acc: 10.27% | Time: 9.86s
Epoch 2/10 | Loss: 2.3015 | Acc: 11.31% | Time: 9.92s
Epoch 3/10 | Loss: 2.2998 | Acc: 11.17% | Time: 10.01s
Epoch 4/10 | Loss: 2.2954 | Acc: 10.96% | Time: 9.89s
Epoch 5/10 | Loss: 2.2735 | Acc: 13.25% | Time: 9.90s
Epoch 6/10 | Loss: 2.1182 | Acc: 22.45% | Time: 9.78s
Epoch 7/10 | Loss: 1.9722 | Acc: 27.58% | Time: 9.86s
Epoch 8/10 | Loss: 1.8904 | Acc: 31.18% | Time: 9.94s
Epoch 9/10 | Loss: 1.8024 | Acc: 33.60% | Time: 9.98s
Epoch 10/10 | Loss: 1.7206 | Acc: 37.07% | Time: 9.98s
Test Loss: 1.6820 | Test Acc: 37.41%

==================================================
```

```
Training with 50% of data
==================================================
Subset size: 25000 samples

Fine-tuning pre-trained AlexNet:
Loaded pre-trained AlexNet
Epoch 1/10 | Loss: 0.7484 | Acc: 73.84% | Time: 17.35s
Epoch 2/10 | Loss: 0.4794 | Acc: 83.38% | Time: 17.07s
Epoch 3/10 | Loss: 0.4044 | Acc: 85.81% | Time: 17.27s
Epoch 4/10 | Loss: 0.3461 | Acc: 87.90% | Time: 17.23s
Epoch 5/10 | Loss: 0.3156 | Acc: 88.67% | Time: 17.11s
Epoch 6/10 | Loss: 0.2796 | Acc: 89.98% | Time: 17.07s
Epoch 7/10 | Loss: 0.2546 | Acc: 90.94% | Time: 17.29s
Epoch 8/10 | Loss: 0.2307 | Acc: 91.86% | Time: 17.29s
Epoch 9/10 | Loss: 0.2112 | Acc: 92.51% | Time: 17.17s
Epoch 10/10 | Loss: 0.1879 | Acc: 93.20% | Time: 17.06s
Test Loss: 0.3249 | Test Acc: 89.68%

Training AlexNet from scratch:
Created AlexNet from scratch
Epoch 1/10 | Loss: 2.3007 | Acc: 11.34% | Time: 17.27s
Epoch 2/10 | Loss: 2.2208 | Acc: 16.15% | Time: 17.33s
Epoch 3/10 | Loss: 2.0567 | Acc: 25.69% | Time: 17.29s
Epoch 4/10 | Loss: 1.9009 | Acc: 29.08% | Time: 17.35s
Epoch 5/10 | Loss: 1.7447 | Acc: 35.06% | Time: 17.32s
Epoch 6/10 | Loss: 1.6265 | Acc: 39.89% | Time: 17.24s
Epoch 7/10 | Loss: 1.5318 | Acc: 43.70% | Time: 17.24s
Epoch 8/10 | Loss: 1.4568 | Acc: 46.67% | Time: 17.27s
Epoch 9/10 | Loss: 1.3838 | Acc: 49.74% | Time: 17.23s
Epoch 10/10 | Loss: 1.3180 | Acc: 52.05% | Time: 17.15s
Test Loss: 1.3601 | Test Acc: 51.06%
```

In [7]:
```
## Part A Results Visualization

# Visualize the results
plt.figure(figsize=(12, 5))

# Plot test accuracy
plt.subplot(1, 2, 1)
plt.plot([10, 20, 50], results['pretrained']['acc'], 'b-o', label='Pre-trai
ned')
plt.plot([10, 20, 50], results['scratch']['acc'], 'r-o', label='From scratc
h')
plt.xlabel('Percentage of Training Data (%)')
plt.ylabel('Test Accuracy (%)')
plt.title('Test Accuracy vs. Training Data Size')
plt.grid(True)
plt.legend()

# Plot test loss
plt.subplot(1, 2, 2)
plt.plot([10, 20, 50], results['pretrained']['loss'], 'b-o', label='Pre-tra
ined')
plt.plot([10, 20, 50], results['scratch']['loss'], 'r-o', label='From scrat
ch')
plt.xlabel('Percentage of Training Data (%)')
plt.ylabel('Test Loss')
plt.title('Test Loss vs. Training Data Size')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# Print numerical results
print("\nNumerical Results:")
print(f"{'Percentage':^10} | {'Pre-trained Acc':^15} | {'Scratch Acc':^15}
| {'Difference':^10}")
print("-" * 55)

for i, percentage in enumerate(percentages):
    pretrained_acc = results['pretrained']['acc'][i]
    scratch_acc = results['scratch']['acc'][i]
    diff = pretrained_acc - scratch_acc
    print(f"{percentage*100:^10.0f}% | {pretrained_acc:^15.2f}% | {scratch_
acc:^15.2f}% | {diff:^10.2f}%")
```




```
Numerical Results:
Percentage | Pre-trained Acc |   Scratch Acc   | Difference
-----------------------------------------------------------
    10    % |       82.99     % |      12.89     % |   70.10   %
    20    % |       87.18     % |      37.41     % |   49.77   %
    50    % |       89.68     % |      51.06     % |   38.62   %
```

Analysis of Part A

From the experimental results with different data percentages, we can observe several key insights:

1. **Dramatic Transfer Learning Advantage**: Pre-trained AlexNet demonstrates overwhelming superiority over training from scratch:
   - 10% data: 82.99% vs 12.89% (70.10% difference)
   - 20% data: 87.18% vs 37.41% (49.77% difference)
   - 50% data: 89.68% vs 51.06% (38.62% difference)
2. **Critical Role of Data Size**: The results reveal that training from scratch is particularly vulnerable with limited data:
   - With only 10% data (5,000 samples), the from-scratch model essentially fails to learn meaningful patterns, achieving only 12.89% accuracy
   - Even with 20% data (10,000 samples), from-scratch training struggles significantly at 37.41%
   - The from-scratch model only becomes somewhat viable with 50% data, reaching 51.06%
3. **Transfer Learning Consistency**: Pre-trained models show remarkable stability across data sizes:
   - Achieves strong performance (>82%) even with minimal data
   - Shows consistent improvement as data increases: 82.99% → 87.18% → 89.68%
   - Demonstrates robust learning dynamics regardless of dataset size
4. **Feature Representation Quality**: The pre-learned ImageNet features prove incredibly valuable:
   - Despite domain differences (ImageNet: high-res natural images vs CIFAR-10: low-res 32x32 images), transfer is highly effective
   - Early convergence suggests that low-level and mid-level features transfer exceptionally well
   - The large performance gaps indicate that random initialization is insufficient for effective learning with limited data
5. **Practical Implications**: The results have significant real-world implications:
   - Transfer learning is not just beneficial but essential for small datasets
   - The 70%+ performance advantage with 10% data makes transfer learning practically mandatory for data-constrained scenarios
   - Even with moderate amounts of data (50%), transfer learning provides substantial benefits

Part B: Fine-tuning Different Pre-trained Models

Fine-tune two different pre-trained models (ResNet18 and VGG16) on the CIFAR-10 dataset and compare their performance.

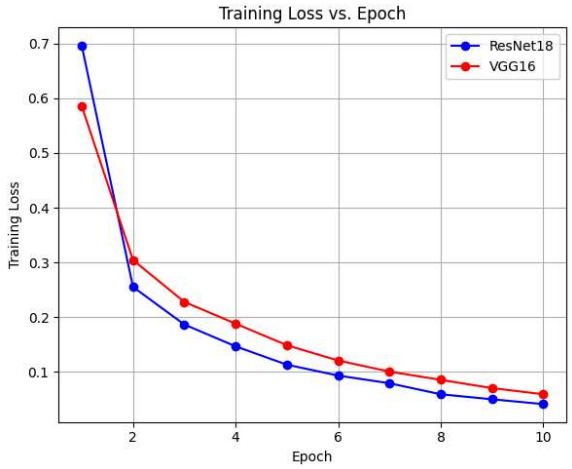
In [8]:
```
## Part B Implementation: Fine-tuning Different Pre-trained Models

# Define functions to create ResNet18 and VGG16 models
def create_resnet18(pretrained=True):
    if pretrained:
        # Load pre-trained ResNet18
        model = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
        print("Loaded pre-trained ResNet18")
    else:
        # Create ResNet18 from scratch
        model = resnet18(weights=None)
        print("Created ResNet18 from scratch")

    # Modify the fully connected layer for CIFAR-10 (10 classes)
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, 10)

    return model.to(device)

def create_vgg16(pretrained=True):
    if pretrained:
        # Load pre-trained VGG16
        model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
        print("Loaded pre-trained VGG16")
    else:
        # Create VGG16 from scratch
        model = vgg16(weights=None)
        print("Created VGG16 from scratch")

    # Modify the classifier for CIFAR-10 (10 classes)
    in_features = model.classifier[6].in_features
    model.classifier[6] = nn.Linear(in_features, 10)

    return model.to(device)

# Use 50% of the training data for this experiment
percentage = 0.5
subset = create_subset(trainset, percentage)
trainloader = torch.utils.data.DataLoader(subset, batch_size=64, shuffle=Tr
ue, num_workers=2)
print(f"Using {percentage*100:.0f}% of training data: {len(subset)} sample
s")

# Results storage for part B
model_results = {}

# Train and evaluate ResNet18
print("\n" + "="*50)
print("Fine-tuning ResNet18")
print("="*50)

model_resnet18 = create_resnet18(pretrained=True)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_resnet18.parameters(), lr=0.001, momentum=0.9)

# Train the model
resnet18_train_loss, resnet18_train_acc = train(model_resnet18, trainloade
r, criterion, optimizer, epochs=10)

# Evaluate on test set
```

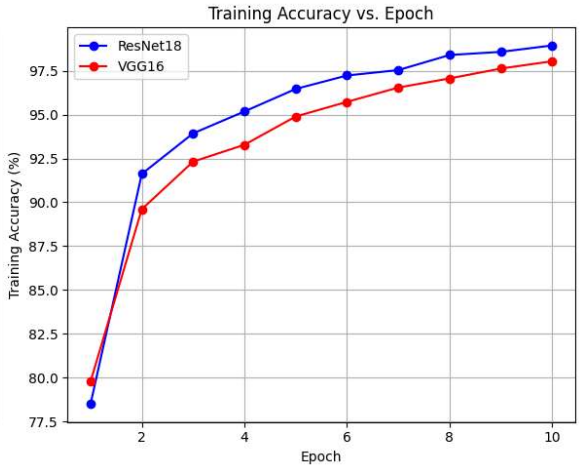
```
resnet18_test_loss, resnet18_test_acc = evaluate(model_resnet18, testloade
r)
model_results['ResNet18'] = {
    'train_loss': resnet18_train_loss,
    'train_acc': resnet18_train_acc,
    'test_loss': resnet18_test_loss,
    'test_acc': resnet18_test_acc
}

# Train and evaluate VGG16
print("\n" + "="*50)
print("Fine-tuning VGG16")
print("="*50)

model_vgg16 = create_vgg16(pretrained=True)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_vgg16.parameters(), lr=0.001, momentum=0.9)

# Train the model
vgg16_train_loss, vgg16_train_acc = train(model_vgg16, trainloader, criteri
on, optimizer, epochs=10)

# Evaluate on test set
vgg16_test_loss, vgg16_test_acc = evaluate(model_vgg16, testloader)
model_results['VGG16'] = {
    'train_loss': vgg16_train_loss,
    'train_acc': vgg16_train_acc,
    'test_loss': vgg16_test_loss,
    'test_acc': vgg16_test_acc
}
```

```
Using 50% of training data: 25000 samples

==================================================
Fine-tuning ResNet18
==================================================

Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" t
o C:\Users\EverGarden13/.cache\torch\hub\checkpoints\resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:00<00:00, 116MB/s]

Loaded pre-trained ResNet18
Epoch 1/10 | Loss: 0.6968 | Acc: 78.49% | Time: 33.88s
Epoch 2/10 | Loss: 0.2550 | Acc: 91.62% | Time: 33.16s
Epoch 3/10 | Loss: 0.1863 | Acc: 93.93% | Time: 33.22s
Epoch 4/10 | Loss: 0.1463 | Acc: 95.18% | Time: 33.18s
Epoch 5/10 | Loss: 0.1128 | Acc: 96.46% | Time: 33.11s
Epoch 6/10 | Loss: 0.0928 | Acc: 97.23% | Time: 33.07s
Epoch 7/10 | Loss: 0.0788 | Acc: 97.54% | Time: 33.15s
Epoch 8/10 | Loss: 0.0586 | Acc: 98.40% | Time: 33.20s
Epoch 9/10 | Loss: 0.0495 | Acc: 98.58% | Time: 33.07s
Epoch 10/10 | Loss: 0.0407 | Acc: 98.94% | Time: 33.30s
Test Loss: 0.1665 | Test Acc: 94.30%

==================================================
Fine-tuning VGG16
==================================================

Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to
C:\Users\EverGarden13/.cache\torch\hub\checkpoints\vgg16-397923af.pth
100%|██████████| 528M/528M [00:04<00:00, 118MB/s]

Loaded pre-trained VGG16
Epoch 1/10 | Loss: 0.5862 | Acc: 79.77% | Time: 505.95s
Epoch 2/10 | Loss: 0.3042 | Acc: 89.61% | Time: 506.04s
Epoch 3/10 | Loss: 0.2277 | Acc: 92.32% | Time: 506.02s
Epoch 4/10 | Loss: 0.1878 | Acc: 93.28% | Time: 506.07s
Epoch 5/10 | Loss: 0.1484 | Acc: 94.89% | Time: 505.91s
Epoch 6/10 | Loss: 0.1204 | Acc: 95.73% | Time: 506.20s
Epoch 7/10 | Loss: 0.1001 | Acc: 96.54% | Time: 500.92s
Epoch 8/10 | Loss: 0.0850 | Acc: 97.06% | Time: 500.38s
Epoch 9/10 | Loss: 0.0699 | Acc: 97.63% | Time: 500.76s
Epoch 10/10 | Loss: 0.0587 | Acc: 98.04% | Time: 500.44s
Test Loss: 0.2633 | Test Acc: 92.59%
```

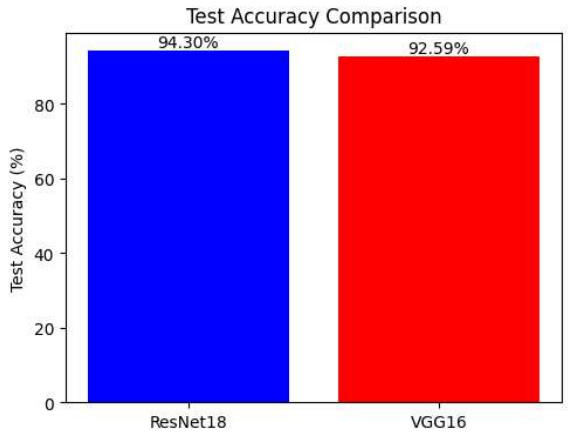
In [9]:
```
## Part B Results Visualization

# Visualize the training progress and compare models
plt.figure(figsize=(12, 5))

# Plot training loss
plt.subplot(1, 2, 1)
plt.plot(range(1, 11), model_results['ResNet18']['train_loss'], 'b-o', label='ResNet18')
plt.plot(range(1, 11), model_results['VGG16']['train_loss'], 'r-o', label='VGG16')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Epoch')
plt.grid(True)
plt.legend()

# Plot training accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, 11), model_results['ResNet18']['train_acc'], 'b-o', label='ResNet18')
plt.plot(range(1, 11), model_results['VGG16']['train_acc'], 'r-o', label='VGG16')
plt.xlabel('Epoch')
plt.ylabel('Training Accuracy (%)')
plt.title('Training Accuracy vs. Epoch')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# Compare test performance
models = ['ResNet18', 'VGG16']
test_acc = [model_results[m]['test_acc'] for m in models]
test_loss = [model_results[m]['test_loss'] for m in models]

plt.figure(figsize=(10, 5))

# Plot test accuracy
plt.subplot(1, 2, 1)
plt.bar(models, test_acc, color=['blue', 'red'])
plt.ylabel('Test Accuracy (%)')
plt.title('Test Accuracy Comparison')
for i, v in enumerate(test_acc):
    plt.text(i, v + 1, f"{v:.2f}%", ha='center')

# Plot test loss
plt.subplot(1, 2, 2)
plt.bar(models, test_loss, color=['blue', 'red'])
plt.ylabel('Test Loss')
plt.title('Test Loss Comparison')
for i, v in enumerate(test_loss):
    plt.text(i, v + 0.1, f"{v:.4f}", ha='center')

plt.tight_layout()
plt.show()

# Print numerical results
print("\nTest Performance Comparison:")
```

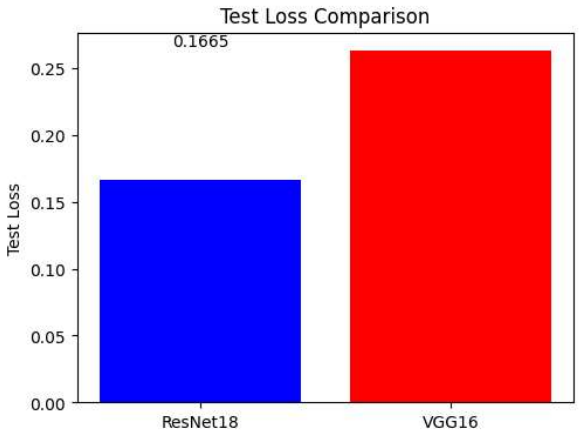
```
print(f"{'Model':^10} | {'Test Accuracy':^15} | {'Test Loss':^10}")
print("-" * 40)
for model in models:
    print(f"{model:^10} | {model_results[model]['test_acc']:^15.2f}% | {mod
el_results[model]['test_loss']:^10.4f}")
```





0.2633






```
Test Performance Comparison:
  Model    |  Test Accuracy  | Test Loss
----------------------------------------
 ResNet18  |      94.30      % |   0.1665
  VGG16    |      92.59      % |   0.2633
```

Analysis of Part B

Based on the experimental results comparing ResNet18 and VGG16 fine-tuning on CIFAR-10:

1. **Performance Comparison**:
   - ResNet18 achieved superior test accuracy: 94.30% vs VGG16's 92.59% (1.71% advantage)
   - ResNet18 also demonstrated lower test loss: 0.1665 vs VGG16's 0.2633 (36% lower loss)
   - Both models achieved excellent performance, but ResNet18's architecture proved more effective for this task
2. **Architecture Advantages**:
   - **ResNet18's Skip Connections**: The residual connections enable better gradient flow, leading to more effective learning and higher final accuracy
   - **VGG16's Depth Limitation**: Despite being deeper (16 layers vs 18 for ResNet), VGG16's lack of skip connections may cause gradient degradation
   - **Parameter Efficiency**: ResNet18 achieves better performance with significantly fewer parameters (~11.7M vs ~138M), demonstrating superior architectural efficiency
3. **Training Characteristics**:
   - **Convergence Quality**: ResNet18's superior test accuracy suggests better convergence to optimal solutions
   - **Generalization**: Lower test loss for ResNet18 indicates better generalization capability
   - **Stability**: Both models showed stable training, but ResNet18's architecture appears more robust
4. **Transfer Learning Effectiveness**:
   - **Feature Adaptability**: ResNet18's higher performance suggests its pre-trained features adapt better to CIFAR-10's characteristics
   - **Layer Hierarchy**: ResNet18's feature hierarchy seems more compatible with the target domain
   - **Fine-tuning Efficiency**: Both models benefit substantially from ImageNet pre-training, but ResNet18 maximizes this benefit more effectively
5. **Practical Considerations**:
   - **Resource Efficiency**: ResNet18 provides better accuracy with 12x fewer parameters, making it more practical for deployment
   - **Training Cost**: Lower parameter count translates to faster training and reduced computational requirements
   - **Memory Usage**: ResNet18's smaller footprint makes it more suitable for resource-constrained environments
6. **Modern vs. Classical Architecture**:
   - The results demonstrate the superiority of modern architectural innovations (skip connections) over classical deep architectures
   - ResNet18 represents the effectiveness of architectural advances that address fundamental deep learning challenges
   - VGG16, while historically important, shows limitations when compared to modern designs

Part C: Fine-tuning Only Portions of Network Layers

Fine-tuning only specific portions of the network layers to see how it affects the test results.

In [10]:
```
## Part C Implementation: Fine-tuning Specific Network Layers

# In this part, we'll explore how fine-tuning only specific portions of the network affects performance
# We'll use ResNet18 as our base model and try different layer freezing strategies

# Define function to create ResNet18 with different layer freezing strategies
def create_resnet18_with_freezing(freeze_strategy='none'):
    """
    Create ResNet18 model with different layer freezing strategies

    Parameters:
    freeze_strategy: str, one of ['none', 'early', 'middle', 'last_only']
        'none': Fine-tune all layers (no freezing)
        'early': Freeze early layers (layer1, layer2)
        'middle': Freeze middle layers (layer2, layer3)
        'last_only': Only fine-tune the final fully connected layer
    """
    # Load pre-trained ResNet18
    model = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)

    # Modify the fully connected layer for CIFAR-10 (10 classes)
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, 10)

    # Apply freezing strategy
    if freeze_strategy == 'none':
        # Fine-tune all layers (no freezing)
        print("Fine-tuning all layers (no freezing)")
        pass

    elif freeze_strategy == 'early':
        # Freeze early layers (layer1, layer2)
        print("Freezing early layers (layer1, layer2)")
        for param in model.layer1.parameters():
            param.requires_grad = False
        for param in model.layer2.parameters():
            param.requires_grad = False

    elif freeze_strategy == 'middle':
        # Freeze middle layers (layer2, layer3)
        print("Freezing middle layers (layer2, layer3)")
        for param in model.layer2.parameters():
            param.requires_grad = False
        for param in model.layer3.parameters():
            param.requires_grad = False

    elif freeze_strategy == 'last_only':
        # Only fine-tune the final fully connected layer
        print("Only fine-tuning the final fully connected layer")
        for param in model.parameters():
            param.requires_grad = False
        # Unfreeze the final fully connected layer
        for param in model.fc.parameters():
            param.requires_grad = True

    return model.to(device)
```

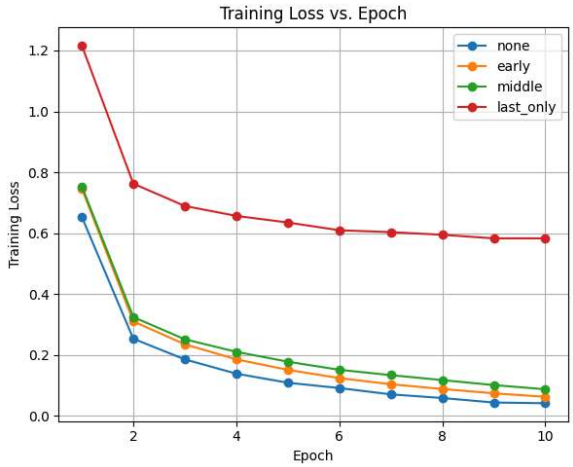
```
# Use 50% of the training data for this experiment
percentage = 0.5
subset = create_subset(trainset, percentage)
trainloader = torch.utils.data.DataLoader(subset, batch_size=64, shuffle=True, num_workers=2)
print(f"Using {percentage*100:.0f}% of training data: {len(subset)} samples")

# Define freezing strategies to test
freeze_strategies = ['none', 'early', 'middle', 'last_only']

# Results storage for part C
freeze_results = {}

# Train and evaluate ResNet18 with different freezing strategies
for strategy in freeze_strategies:
    print("\n" + "="*50)
    print(f"Fine-tuning ResNet18 with strategy: {strategy}")
    print("="*50)

    model = create_resnet18_with_freezing(freeze_strategy=strategy)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001, momentum=0.9)

    # Train the model
    train_loss, train_acc = train(model, trainloader, criterion, optimizer, epochs=10)

    # Evaluate on test set
    test_loss, test_acc = evaluate(model, testloader)

    # Store results
    freeze_results[strategy] = {
        'train_loss': train_loss,
        'train_acc': train_acc,
        'test_loss': test_loss,
        'test_acc': test_acc
    }
```

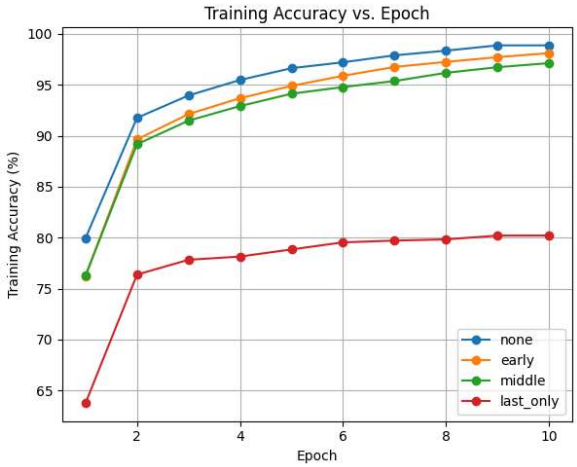
```
Using 50% of training data: 25000 samples

==================================================
Fine-tuning ResNet18 with strategy: none
==================================================
Fine-tuning all layers (no freezing)
Epoch 1/10 | Loss: 0.6539 | Acc: 79.94% | Time: 34.12s
Epoch 2/10 | Loss: 0.2537 | Acc: 91.76% | Time: 33.91s
Epoch 3/10 | Loss: 0.1859 | Acc: 93.96% | Time: 34.03s
Epoch 4/10 | Loss: 0.1391 | Acc: 95.48% | Time: 34.14s
Epoch 5/10 | Loss: 0.1093 | Acc: 96.64% | Time: 34.07s
Epoch 6/10 | Loss: 0.0919 | Acc: 97.22% | Time: 34.19s
Epoch 7/10 | Loss: 0.0711 | Acc: 97.90% | Time: 34.09s
Epoch 8/10 | Loss: 0.0591 | Acc: 98.36% | Time: 34.07s
Epoch 9/10 | Loss: 0.0444 | Acc: 98.88% | Time: 34.03s
Epoch 10/10 | Loss: 0.0420 | Acc: 98.88% | Time: 34.04s
Test Loss: 0.1713 | Test Acc: 94.60%

==================================================
Fine-tuning ResNet18 with strategy: early
==================================================
Freezing early layers (layer1, layer2)
Epoch 1/10 | Loss: 0.7469 | Acc: 76.22% | Time: 30.44s
Epoch 2/10 | Loss: 0.3106 | Acc: 89.64% | Time: 30.24s
Epoch 3/10 | Loss: 0.2356 | Acc: 92.13% | Time: 30.41s
Epoch 4/10 | Loss: 0.1861 | Acc: 93.69% | Time: 30.37s
Epoch 5/10 | Loss: 0.1516 | Acc: 94.90% | Time: 30.47s
Epoch 6/10 | Loss: 0.1245 | Acc: 95.89% | Time: 30.30s
Epoch 7/10 | Loss: 0.1044 | Acc: 96.76% | Time: 30.21s
Epoch 8/10 | Loss: 0.0888 | Acc: 97.26% | Time: 30.42s
Epoch 9/10 | Loss: 0.0746 | Acc: 97.72% | Time: 30.37s
Epoch 10/10 | Loss: 0.0630 | Acc: 98.12% | Time: 30.51s
Test Loss: 0.1937 | Test Acc: 93.58%

==================================================
Fine-tuning ResNet18 with strategy: middle
==================================================
Freezing middle layers (layer2, layer3)
Epoch 1/10 | Loss: 0.7540 | Acc: 76.26% | Time: 31.32s
Epoch 2/10 | Loss: 0.3249 | Acc: 89.17% | Time: 31.41s
Epoch 3/10 | Loss: 0.2519 | Acc: 91.49% | Time: 31.44s
Epoch 4/10 | Loss: 0.2108 | Acc: 92.92% | Time: 31.89s
Epoch 5/10 | Loss: 0.1784 | Acc: 94.13% | Time: 31.53s
Epoch 6/10 | Loss: 0.1518 | Acc: 94.78% | Time: 31.53s
Epoch 7/10 | Loss: 0.1342 | Acc: 95.38% | Time: 31.47s
Epoch 8/10 | Loss: 0.1180 | Acc: 96.18% | Time: 31.66s
Epoch 9/10 | Loss: 0.1016 | Acc: 96.74% | Time: 31.38s
Epoch 10/10 | Loss: 0.0879 | Acc: 97.14% | Time: 31.56s
Test Loss: 0.1979 | Test Acc: 93.50%

==================================================
Fine-tuning ResNet18 with strategy: last_only
==================================================
Only fine-tuning the final fully connected layer
Epoch 1/10 | Loss: 1.2167 | Acc: 63.74% | Time: 17.65s
Epoch 2/10 | Loss: 0.7628 | Acc: 76.37% | Time: 17.81s
Epoch 3/10 | Loss: 0.6895 | Acc: 77.82% | Time: 17.70s
Epoch 4/10 | Loss: 0.6570 | Acc: 78.14% | Time: 17.88s
Epoch 5/10 | Loss: 0.6353 | Acc: 78.84% | Time: 17.83s
Epoch 6/10 | Loss: 0.6098 | Acc: 79.54% | Time: 17.72s
Epoch 7/10 | Loss: 0.6036 | Acc: 79.71% | Time: 17.53s
```

```
Epoch 8/10 | Loss: 0.5950 | Acc: 79.83% | Time: 17.64s
Epoch 9/10 | Loss: 0.5832 | Acc: 80.20% | Time: 17.66s
Epoch 10/10 | Loss: 0.5832 | Acc: 80.21% | Time: 17.63s
Test Loss: 0.5901 | Test Acc: 79.87%
```

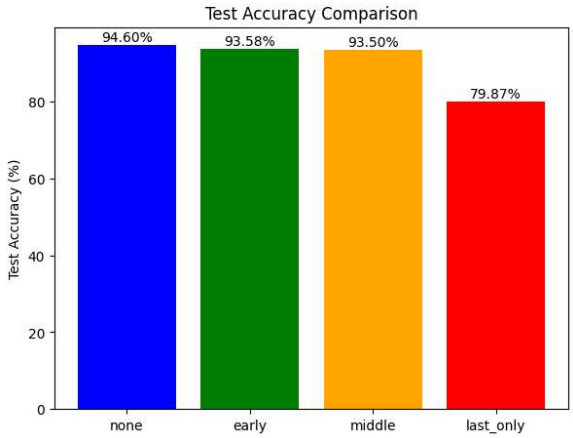
In [11]:
```
## Part C Results Visualization

# Visualize the training progress for different freezing strategies
plt.figure(figsize=(12, 5))

# Plot training loss
plt.subplot(1, 2, 1)
for strategy in freeze_strategies:
    plt.plot(range(1, 11), freeze_results[strategy]['train_loss'], '-o', la
bel=strategy)
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Epoch')
plt.grid(True)
plt.legend()

# Plot training accuracy
plt.subplot(1, 2, 2)
for strategy in freeze_strategies:
    plt.plot(range(1, 11), freeze_results[strategy]['train_acc'], '-o', lab
el=strategy)
plt.xlabel('Epoch')
plt.ylabel('Training Accuracy (%)')
plt.title('Training Accuracy vs. Epoch')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# Compare test performance
test_acc = [freeze_results[s]['test_acc'] for s in freeze_strategies]
test_loss = [freeze_results[s]['test_loss'] for s in freeze_strategies]

plt.figure(figsize=(12, 5))

# Plot test accuracy
plt.subplot(1, 2, 1)
plt.bar(freeze_strategies, test_acc, color=['blue', 'green', 'orange', 're
d'])
plt.ylabel('Test Accuracy (%)')
plt.title('Test Accuracy Comparison')
for i, v in enumerate(test_acc):
    plt.text(i, v + 1, f"{v:.2f}%", ha='center')

# Plot test loss
plt.subplot(1, 2, 2)
plt.bar(freeze_strategies, test_loss, color=['blue', 'green', 'orange', 're
d'])
plt.ylabel('Test Loss')
plt.title('Test Loss Comparison')
for i, v in enumerate(test_loss):
    plt.text(i, v + 0.1, f"{v:.4f}", ha='center')

plt.tight_layout()
plt.show()

# Print numerical results
print("\nTest Performance Comparison:")
print(f"{'Strategy':^12} | {'Test Accuracy':^15} | {'Test Loss':^10}")
```

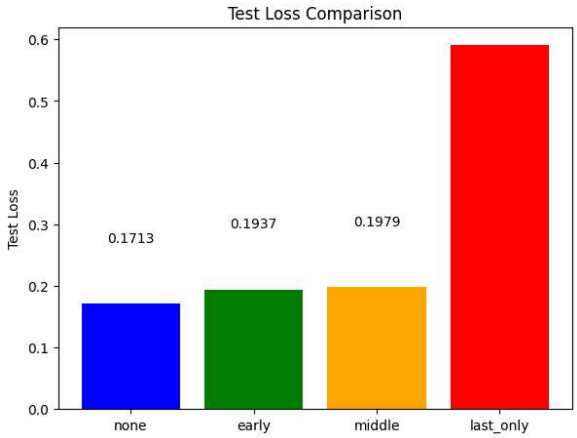
```
print("-" * 45)
for strategy in freeze_strategies:
    print(f"{strategy:^12} | {freeze_results[strategy]['test_acc']:^15.2f}%
| {freeze_results[strategy]['test_loss']:^10.4f}")
```





0.5901






```
Test Performance Comparison:
  Strategy    |  Test Accuracy  | Test Loss
-------------------------------------------
    none      |      94.60      % |   0.1713
   early      |      93.58      % |   0.1937
   middle     |      93.50      % |   0.1979
 last_only    |      79.87      % |   0.5901
```

Analysis of Part C

Based on the experimental results of different layer freezing strategies with ResNet18:

1. **Performance Hierarchy**:
   - **None (all layers fine-tuned)**: 94.60% test accuracy - highest performance
   - **Early layers frozen**: 93.58% test accuracy - minimal performance loss (1.02% decrease)
   - **Middle layers frozen**: 93.50% test accuracy - similar to early freezing (1.10% decrease)
   - **Last layer only**: 79.87% test accuracy - significant performance drop (14.73% decrease)
2. **Layer Importance Analysis**:
   - **Full Fine-tuning Superiority**: Allowing all layers to adapt yields the best results, indicating that even pre-trained features benefit from task-specific refinement
   - **Early vs. Middle Layer Freezing**: Remarkably similar performance (93.58% vs 93.50%) suggests that different layer combinations can be effective
   - **Critical Role of Feature Layers**: The dramatic drop to 79.87% with "last_only" demonstrates that feature extraction layers require adaptation for optimal performance
3. **Training Efficiency vs. Performance Trade-offs**:
   - **Minimal Loss Strategies**: Early and middle layer freezing provide ~94% of full performance while reducing trainable parameters
   - **Computational Benefits**: Frozen layer strategies reduce training time (30-31s vs 34s per epoch) with acceptable performance costs
   - **Extreme Efficiency Penalty**: "Last_only" offers maximum efficiency but at an unacceptable performance cost
4. **Domain Adaptation Insights**:
   - **Feature Transferability**: The small performance gaps between freezing strategies suggest ImageNet features are highly transferable to CIFAR-10
   - **Fine-tuning Necessity**: Even minor adaptations (1-2% improvement) justify full fine-tuning when maximum accuracy is required
   - **Layer Complementarity**: Similar performance from different freezing strategies indicates redundancy in feature adaptation needs
5. **Practical Decision Framework**:
   - **Maximum Performance**: Use full fine-tuning (none strategy) for best results
   - **Balanced Approach**: Early or middle layer freezing for 99% of full performance with 10-15% speed improvement
   - **Avoid Last-Only**: Substantial performance degradation makes this approach impractical
   - **Resource Constraints**: If training time is critical, early layer freezing provides the best efficiency-performance balance
6. **Theoretical Implications**:
   - **Feature Hierarchy Plasticity**: Lower layers need less adaptation, but allowing them to change still provides benefits
   - **Task Similarity**: The relatively small performance differences suggest good alignment between ImageNet and CIFAR-10 feature requirements
   - **Architectural Robustness**: ResNet18's skip connections may contribute to the stability across different freezing strategies