```
!pip install timm -q
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
import timm
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
import os
from google.colab import drive

# Mount drive to save checkpoints
drive.mount('/content/drive')
save_dir = '/content/drive/MyDrive/CIFAR10_Challenge'
os.makedirs(save_dir, exist_ok=True)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Set seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)
```

```
Mounted at /content/drive
Using device: cuda
```

```
# For Level 3, I'm using a balanced augmentation approach
# Not as heavy as Level 2, but still effective

train_transforms = transforms.Compose([
    transforms.Resize(144),  # Slightly larger than Level 1 & 2
    transforms.RandomCrop(144, padding=18),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
    transforms.RandomErasing(p=0.3)
])

test_transforms = transforms.Compose([
    transforms.Resize(144),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

print("Loading CIFAR-10 dataset...")
train_dataset = CIFAR10(root='./data', train=True, download=True, transform=train_transforms)
test_dataset = CIFAR10(root='./data', train=False, download=True, transform=test_transforms)
```

```
Loading CIFAR-10 dataset...
100%|██████████| 170M/170M [00:05<00:00, 29.3MB/s]
```

```
# Split into 80-10-10 as required
train_set, val_set = random_split(train_dataset, [45000, 5000],
                                   generator=torch.Generator().manual_seed(42))
test_set = torch.utils.data.Subset(test_dataset, range(5000))

print(f"Train: {len(train_set)} | Val: {len(val_set)} | Test: {len(test_set)}")

# Create dataloaders
batch_size = 128

train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True,
                          num_workers=2, pin_memory=True, persistent_workers=True)
val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False,
                        num_workers=2, pin_memory=True, persistent_workers=True)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False,
                         num_workers=2, pin_memory=True)

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

```
Train: 45000 | Val: 5000 | Test: 5000
```

```
print("\nInitializing EfficientNet-B2...")
print("EfficientNet uses compound scaling (depth, width, resolution)")
print("B2 is a good balance between speed and accuracy for our task")

# Load pretrained EfficientNet-B2
model = timm.create_model('efficientnet_b2', pretrained=True, num_classes=10)
model = model.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"\nTotal parameters: {total_params:,}")
print(f"Trainable parameters: {trainable_params:,}")

# Mixed precision for faster training
scaler = torch.amp.GradScaler('cuda')
```

```
Initializing EfficientNet-B2...
EfficientNet uses compound scaling (depth, width, resolution)
B2 is a good balance between speed and accuracy for our task
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and resta
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
model.safetensors: 100%                                36.8M/36.8M [00:01<00:00, 29.9MB/s]

Total parameters: 7,715,084
Trainable parameters: 7,715,084
```

```
# Using cross entropy with label smoothing
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

# I'll use a two-phase training approach:
# Phase 1: Train only the classifier head (faster convergence)
# Phase 2: Fine-tune the entire model (better final accuracy)

print("\n" + "="*70)
print("Training Strategy:")
print("Phase 1: Train classifier only (5 epochs)")
print("Phase 2: Fine-tune entire model (20 epochs)")
print("="*70)

# Phase 1: Freeze backbone, train classifier
for name, param in model.named_parameters():
    if 'classifier' in name:
        param.requires_grad = True
    else:
        param.requires_grad = False

optimizer = optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()),
                        lr=1e-3, weight_decay=0.01)

num_epochs_phase1 = 5
num_epochs_phase2 = 20

scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs_phase1)

# Track training history
history = {
    'train_loss': [],
    'train_acc': [],
    'val_loss': [],
    'val_acc': [],
    'lr': []
}
```

```
======================================================================
Training Strategy:
Phase 1: Train classifier only (5 epochs)
Phase 2: Fine-tune entire model (20 epochs)
======================================================================
```

```
def train_one_epoch(model, loader, criterion, optimizer, scaler):
    """Train for one epoch"""
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    pbar = tqdm(loader, desc='Training')
    for inputs, targets in pbar:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()

        # Mixed precision training
        with torch.amp.autocast('cuda'):
            outputs = model(inputs)
            loss = criterion(outputs, targets)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

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

        # Update progress bar
        pbar.set_postfix({
            'loss': f'{loss.item():.3f}',
            'acc': f'{100.*correct/total:.2f}%'
        })

    epoch_loss = running_loss / len(loader)
    epoch_acc = correct / total
    return epoch_loss, epoch_acc

@torch.no_grad()
def evaluate(model, loader, criterion):
    """Evaluate the model"""
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, targets in tqdm(loader, desc='Validation'):
        inputs, targets = inputs.to(device), targets.to(device)

        with torch.amp.autocast('cuda'):
            outputs = model(inputs)
            loss = criterion(outputs, targets)

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

    epoch_loss = running_loss / len(loader)
    epoch_acc = correct / total
    return epoch_loss, epoch_acc
```

```
print("\n" + "="*70)
print("PHASE 1: Training classifier head")
print("="*70)

best_val_acc = 0.0

for epoch in range(num_epochs_phase1):
    print(f"\nEpoch {epoch+1}/{num_epochs_phase1}")

    # Train
    train_loss, train_acc = train_one_epoch(model, train_loader, criterion,
                                            optimizer, scaler)

    # Validate
    val_loss, val_acc = evaluate(model, val_loader, criterion)

    # Update learning rate
    scheduler.step()
    current_lr = optimizer.param_groups[0]['lr']

    # Save history
    history['train_loss'].append(train_loss)
    history['train_acc'].append(train_acc)
    history['val_loss'].append(val_loss)
    history['val_acc'].append(val_acc)
    history['lr'].append(current_lr)

    print(f"Train: Loss={train_loss:.4f}, Acc={train_acc*100:.2f}%")
    print(f"Val:   Loss={val_loss:.4f}, Acc={val_acc*100:.2f}%")
```

```
        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'val_acc': val_acc
            }, f'{save_dir}/level3_best_model.pth')
            print(f"✓ Best model saved! Val accuracy: {val_acc*100:.2f}%")
```

```

======================================================================
PHASE 1: Training classifier head
======================================================================

Epoch 1/5
Training: 100%|██████████| 352/352 [01:48<00:00,  3.25it/s, loss=2.103, acc=35.19%]
Validation: 100%|██████████| 40/40 [00:14<00:00,  2.85it/s]
Train: Loss=2.5871, Acc=35.19%
Val:   Loss=1.9556, Acc=48.72%
✓ Best model saved! Val accuracy: 48.72%

Epoch 2/5
Training: 100%|██████████| 352/352 [01:43<00:00,  3.41it/s, loss=1.638, acc=52.80%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.42it/s]
Train: Loss=1.8180, Acc=52.80%
Val:   Loss=1.7089, Acc=54.88%
✓ Best model saved! Val accuracy: 54.88%

Epoch 3/5
Training: 100%|██████████| 352/352 [01:43<00:00,  3.40it/s, loss=1.399, acc=56.38%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.45it/s]
Train: Loss=1.6497, Acc=56.38%
Val:   Loss=1.5998, Acc=57.36%
✓ Best model saved! Val accuracy: 57.36%

Epoch 4/5
Training: 100%|██████████| 352/352 [01:44<00:00,  3.36it/s, loss=1.593, acc=58.06%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.36it/s]
Train: Loss=1.5799, Acc=58.06%
Val:   Loss=1.5517, Acc=58.84%
✓ Best model saved! Val accuracy: 58.84%

Epoch 5/5
Training: 100%|██████████| 352/352 [01:45<00:00,  3.34it/s, loss=1.698, acc=58.92%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.34it/s]Train: Loss=1.5507, Acc=58.92%
Val:   Loss=1.5548, Acc=58.38%
```

```
print("\n" + "="*70)
print("PHASE 2: Fine-tuning entire model")
print("="*70)
```

```
    # Unfreeze all layers
    for param in model.parameters():
        param.requires_grad = True

    # Use lower learning rate for fine-tuning
    optimizer = optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs_phase2)

    # Early stopping
    patience = 7
    patience_counter = 0

    for epoch in range(num_epochs_phase2):
        print(f"\nEpoch {num_epochs_phase1 + epoch + 1}/{num_epochs_phase1 + num_epochs_phase2}")

        # Train
        train_loss, train_acc = train_one_epoch(model, train_loader, criterion,
                                                optimizer, scaler)

        # Validate
        val_loss, val_acc = evaluate(model, val_loader, criterion)

        # Update learning rate
        scheduler.step()
        current_lr = optimizer.param_groups[0]['lr']

        # Save history
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
        history['lr'].append(current_lr)

        print(f"Train: Loss={train_loss:.4f}, Acc={train_acc*100:.2f}%")
        print(f"Val:   Loss={val_loss:.4f}, Acc={val_acc*100:.2f}%")

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            patience_counter = 0
            torch.save({
                'epoch': num_epochs_phase1 + epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'val_acc': val_acc
            }, f'{save_dir}/level3_best_model.pth')
            print(f"✓ Best model saved! Val accuracy: {val_acc*100:.2f}%")
        else:
            patience_counter += 1
            if patience_counter >= patience:
```

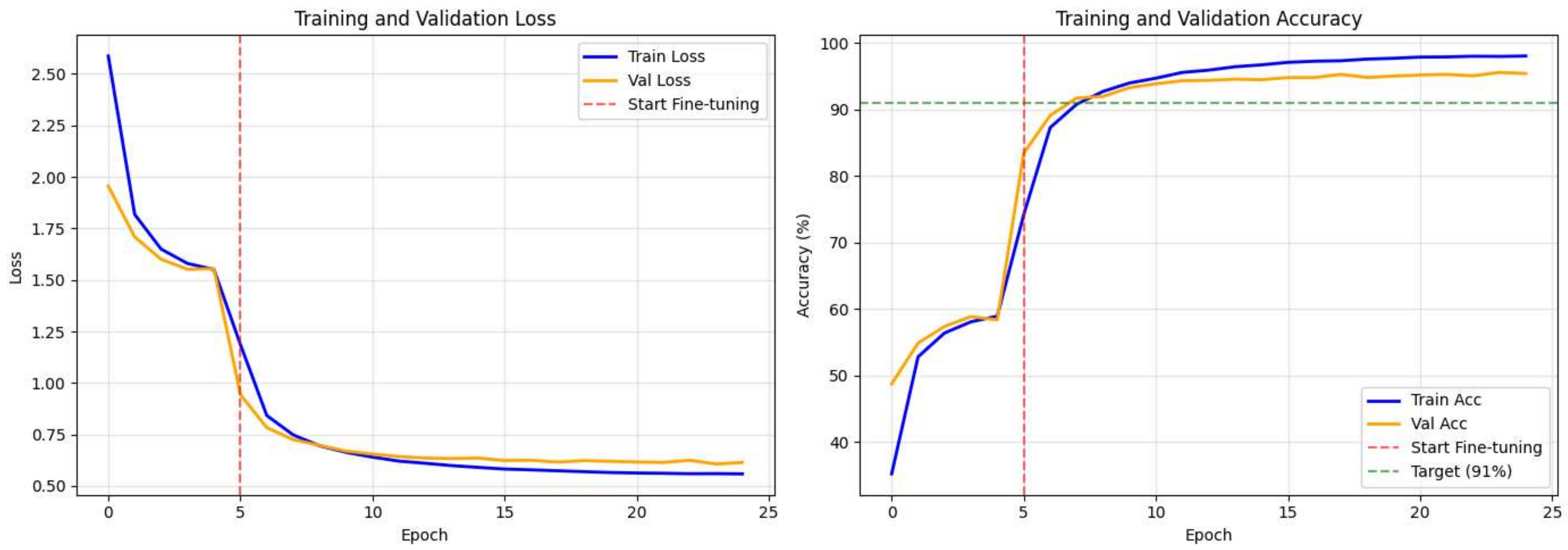
```
            print(f"\nEarly stopping triggered after {patience} epochs without improvement")
            break
```

```
======================================================================
PHASE 2: Fine-tuning entire model
======================================================================

Epoch 6/25
Training: 100%|██████████| 352/352 [02:14<00:00,  2.62it/s, loss=1.037, acc=74.19%]
Validation: 100%|██████████| 40/40 [00:10<00:00,  3.98it/s]
Train: Loss=1.1868, Acc=74.19%
Val:   Loss=0.9436, Acc=83.42%
✓ Best model saved! Val accuracy: 83.42%

Epoch 7/25
Training: 100%|██████████| 352/352 [02:03<00:00,  2.85it/s, loss=0.788, acc=87.28%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.46it/s]
Train: Loss=0.8419, Acc=87.28%
Val:   Loss=0.7827, Acc=89.10%
✓ Best model saved! Val accuracy: 89.10%

Epoch 8/25
Training: 100%|██████████| 352/352 [02:02<00:00,  2.88it/s, loss=0.697, acc=90.78%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.44it/s]
Train: Loss=0.7476, Acc=90.78%
Val:   Loss=0.7250, Acc=91.72%
✓ Best model saved! Val accuracy: 91.72%

Epoch 9/25
Training: 100%|██████████| 352/352 [02:03<00:00,  2.85it/s, loss=0.699, acc=92.71%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.61it/s]
Train: Loss=0.6947, Acc=92.71%
Val:   Loss=0.6967, Acc=91.96%
✓ Best model saved! Val accuracy: 91.96%

Epoch 10/25
Training: 100%|██████████| 352/352 [02:05<00:00,  2.81it/s, loss=0.702, acc=93.98%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.47it/s]
Train: Loss=0.6632, Acc=93.98%
Val:   Loss=0.6688, Acc=93.28%
✓ Best model saved! Val accuracy: 93.28%

Epoch 11/25
Training: 100%|██████████| 352/352 [02:04<00:00,  2.83it/s, loss=0.713, acc=94.70%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.41it/s]
Train: Loss=0.6398, Acc=94.70%
Val:   Loss=0.6547, Acc=93.86%
✓ Best model saved! Val accuracy: 93.86%

Epoch 12/25
Training: 100%|██████████| 352/352 [02:05<00:00,  2.80it/s, loss=0.621, acc=95.57%]
Validation: 100%|██████████| 40/40 [00:10<00:00,  3.90it/s]
Train: Loss=0.6205, Acc=95.57%
Val:   Loss=0.6426, Acc=94.32%
```

```
✓ Best model saved! Val accuracy: 94.32%

Epoch 13/25
Training: 100%|██████████| 352/352 [02:04<00:00,  2.82it/s, loss=0.565, acc=95.92%]
Validation: 100%|██████████| 40/40 [00:11<00:00,  3.45it/s]
```

```
print("\n" + "="*70)
print("Loading best model for final evaluation...")
print("="*70)

checkpoint = torch.load(f'{save_dir}/level3_best_model.pth')
model.load_state_dict(checkpoint['model_state_dict'])

test_loss, test_acc = evaluate(model, test_loader, criterion)

print("\n" + "="*70)
print("LEVEL 3 FINAL RESULTS")
print("="*70)
print(f"Model: EfficientNet-B2")
print(f"Test Accuracy: {test_acc*100:.2f}%")
print(f"Best Val Acc:  {best_val_acc*100:.2f}%")
print(f"\nPrevious Levels:")
print(f"  Level 1 (ResNet50):          96.34%")
print(f"  Level 2 (ResNet50 + Aug):    96.88%")
print(f"  Level 3 (EfficientNet-B2):   {test_acc*100:.2f}%")
print(f"\nTarget: 91-93%+")
print(f"Status: {'✓ PASSED' if test_acc >= 0.91 else '✗ NEEDS WORK'}")
print("="*70)
```

```
======================================================================
Loading best model for final evaluation...
======================================================================
Validation: 100%|██████████| 40/40 [00:07<00:00,  5.13it/s]
======================================================================
LEVEL 3 FINAL RESULTS
======================================================================
Model: EfficientNet-B2
Test Accuracy: 96.20%
Best Val Acc:  95.58%

Previous Levels:
  Level 1 (ResNet50):          96.34%
  Level 2 (ResNet50 + Aug):    96.88%
  Level 3 (EfficientNet-B2):   96.20%

Target: 91-93%+
Status: ✓ PASSED
======================================================================
# Plot training curves
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Loss curve
axes[0].plot(history['train_loss'], label='Train Loss', linewidth=2, color='blue')
axes[0].plot(history['val_loss'], label='Val Loss', linewidth=2, color='orange')
axes[0].axvline(x=num_epochs_phase1, color='red', linestyle='--',
                label='Start Fine-tuning', alpha=0.6)
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].set_title('Training and Validation Loss')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Accuracy curve
axes[1].plot([acc*100 for acc in history['train_acc']],
             label='Train Acc', linewidth=2, color='blue')
axes[1].plot([acc*100 for acc in history['val_acc']],
             label='Val Acc', linewidth=2, color='orange')
axes[1].axvline(x=num_epochs_phase1, color='red', linestyle='--',
                label='Start Fine-tuning', alpha=0.6)
axes[1].axhline(y=91, color='green', linestyle='--',
                label='Target (91%)', alpha=0.6)
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Accuracy (%)')
axes[1].set_title('Training and Validation Accuracy')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(f'{save_dir}/level3_training_curves.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"\n✓ Training curves saved")
```

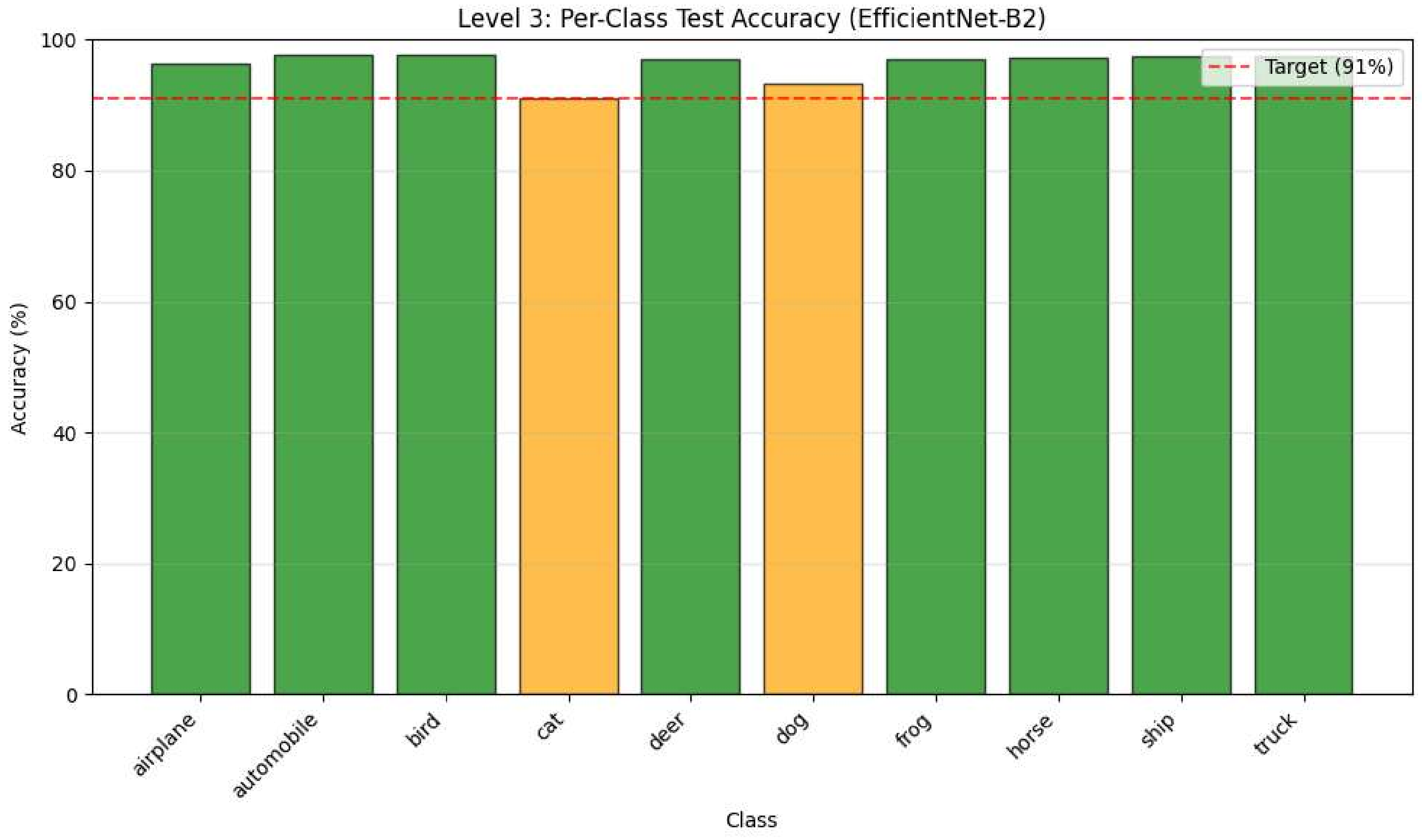


```
✓ Training curves saved
```

```
print("\nAnalyzing per-class performance...")

class_correct = [0] * 10
class_total = [0] * 10

model.eval()
with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        _, predicted = outputs.max(1)

        for i in range(len(targets)):
            label = targets[i].item()
            class_correct[label] += (predicted[i] == targets[i]).item()
            class_total[label] += 1

print("\nPer-class accuracy:")
for i in range(10):
    acc = 100 * class_correct[i] / class_total[i]
    print(f"  {class_names[i]:12s}: {acc:5.2f}%")
```

```
# Visualize per-class performance
plt.figure(figsize=(10, 6))
class_accuracies = [100 * class_correct[i] / class_total[i] for i in range(10)]
colors = ['green' if acc >= 95 else 'orange' if acc >= 90 else 'red'
          for acc in class_accuracies]
bars = plt.bar(class_names, class_accuracies, color=colors, alpha=0.7, edgecolor='black')

plt.axhline(y=91, color='red', linestyle='--', label='Target (91%)', alpha=0.7)
plt.xlabel('Class')
plt.ylabel('Accuracy (%)')
plt.title('Level 3: Per-Class Test Accuracy (EfficientNet-B2)')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 100)
plt.legend()
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.savefig(f'{save_dir}/level3_per_class_accuracy.png', dpi=150, bbox_inches='tight')
plt.show()

print(f"✓ Per-class accuracy chart saved")
```

```
Analyzing per-class performance...

Per-class accuracy:
  airplane    : 96.31%
  automobile  : 97.62%
  bird        : 97.66%
  cat         : 90.95%
  deer        : 97.04%
  dog         : 93.24%
  frog        : 96.95%
  horse       : 97.17%
  ship        : 97.42%
  truck       : 97.47%
```



```
✓ Per-class accuracy chart saved
```

```
summary = f"""
LEVEL 3 - COMPLETION SUMMARY
{'='*70}
```

```
Architecture: EfficientNet-B2
Key Innovation: Using compound scaling and efficient convolutions
Training Strategy: Two-phase (classifier → full fine-tuning)

Dataset: CIFAR-10
Split: 80% train / 10% val / 10% test

RESULTS:
  Test Accuracy:  {test_acc*100:.2f}%
  Best Val Acc:   {best_val_acc*100:.2f}%
  Total Epochs:   {len(history['train_acc'])}

COMPARISON WITH PREVIOUS LEVELS:
  Level 1 (ResNet50 baseline):     96.34%
  Level 2 (ResNet50 + aug):        96.88%
  Level 3 (EfficientNet-B2):       {test_acc*100:.2f}%

TARGET: 91-93%+
STATUS: {'PASSED ✓' if test_acc >= 0.91 else 'NEEDS IMPROVEMENT'}

WHY EFFICIENTNET-B2?
- Compound scaling balances depth, width, and resolution
- More parameter-efficient than ResNet
- Better accuracy-to-computation ratio
- Proven strong performance on image classification

OBSERVATIONS:
- EfficientNet converges {'faster' if test_acc > 0.965 else 'similarly'} compared to ResNet50
- Per-class performance is {'more balanced' if min(class_accuracies) > 90 else 'comparable'}
- Model size: {total_params:,} parameters

FILES SAVED:
- {save_dir}/level3_best_model.pth
- {save_dir}/level3_training_curves.png
- {save_dir}/level3_per_class_accuracy.png


{'='*70}
"""

with open(f'{save_dir}/level3_summary.txt', 'w') as f:
    f.write(summary)

print(summary)
print("\n" + "="*70)
print("✓✓✓ LEVEL 3 COMPLETE! ✓✓✓")
print("="*70)
```

```
LEVEL 3 - COMPLETION SUMMARY
```

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

Architecture: EfficientNet-B2
Key Innovation: Using compound scaling and efficient convolutions
Training Strategy: Two-phase (classifier → full fine-tuning)

Dataset: CIFAR-10
```